1993

An object orient program specification for a mobile robot motion control language

Grim, Carl Joseph

Monterey, California. Naval Postgraduate School

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
An Object Oriented Program Specification for the Mobile Robot Motion Control Language

12. PERSONAL AUTHOR(S)
Carl Joseph Grim

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 07/90 TO 03/93	1993, March 25	119

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	MML, OOPS-MML

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Yamabico Research Group at the Naval Postgraduate School is actively pursuing improvements in design and implementation of applications for it's family of autonomous mobile robots. This paper describes a new high level language for controlling the Yamabico-11, surnamed OOPS-MML (Object-Oriented Program Specification for a Mobile robot Motion control Language). Conceptual goals included a user friendly, high level interface coupled with a very abstract, efficient and compartmentalized architecture to employ a path planning and tracking application developed at NPS. The result is a robust and flexible robot control system that is intended to be implemented and employed onboard the Yamabico-11.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Yutaka Kanayama	22b. TELEPHONE (Include Area Code): (408) 646-2174	22c. OFFICE SYMBOL: CS

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

T260460

Approved for public release; distribution is unlimited

An Object Orient Program Specification For A Mobile Robot Motion Control Language

by

Carl J. Grim

Lieutenant, United States Naval Reserve

B. A. S. in Business Administration, Capital University, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1993

ABSTRACT

The Yamabico Research Group at the Naval Postgraduate School is actively pursuing improvements in design and implementation of applications for it's family of autonomous mobile robots. This paper describes a new high level language for controlling the Yamabico-11, surnamed OOPS-MML (Object-Oriented Program Specification for a Mobile robot Motion control Language). Conceptual goals included a user friendly, high level interface coupled with an abstract, efficient and compartmentalized architecture to employ a path tracking and motion control application developed at NPS. The result is a robust and flexible robot control system that is intended to be implemented and employed onboard the Yamabico-11.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

The goal of any robot is to adequately perform the task for which it was designed. The "Yamabico" family of autonomous mobile robots are ultimately being designed to operate on time scales of its human counterpart and encompass the intelligence to navigate in an actively dynamic environment.

To accomplish this, Yamabico must first be able to effectively create and transit a nominal path through a static environment. Secondly, it must possess the intelligence and capabilities to exhibit robust behavior in generating an acceptable path within a variety of static environments ranging in degrees of adversity.

The current state of the Yamabico-11's design is one of being fully autonomous. The robot's intelligence consists primarily of two behaviors. The first, is being able to follow a sequence of given postures represented by (x, y, θ). The second, is being able to sense an object by sonar, such as a wall, and follow it. These are accomplished by the use of the Mobile robot Motion control Language, or MML for short. This MML library includes functions for path planning, path generation, maintenance of sonar data, motion control and tracking.

Recent research conducted at NPS has produced the basis for a simple technique to provide for smooth path planning [1]. This technique greatly enhances robotic motion and intelligence. Allowing the capabilities for Yamabico to exhibit the robust behavior needed to navigate a variety of static environments and ultimately be able to survive in an actively dynamic environment. At this point, the decision was made to pursue the design and implementation of a new MML library to be incorporated into Yamabico-11 to utilize this new technique.

A. FUNCTIONAL GOALS

Yamabico has proven to be an invaluable tool for research in basic robotic courses offered here at NPS, since it affords a student with the relatively simple means for conducting research. This is due in part to its physically characteristics of operating only within two dimensions and being able to be easily deployed by one person. Mainly though, it is due to the

simplistic nature and compactness of it's high level language MML which the student utilizes to write user programs for controlling robot motion and testing theories. With this in mind, any new language must follow this same pretext.

Yamabico has proven to be an invaluable asset in the area of advanced robotic research here at NPS mainly for two reasons. The first, by allowing the opportunity to have "hands on" of a robot from the beginning, sparks the enthusiasm of students and promulgates advanced studies. The second is not from Yamabico's capabilities but it's lack of capabilities, which results in advanced study to overcome or create them. With these in mind, any new language must include advancements due to both refinements and creativity in order to hopefully spawn off further studies.

Thus, the functional goal for OOPS-MML is that it be simple yet powerful to allow both spectrums to adequately conduct research. To make the language simple, it is paramount that we offer a small set of clear, concise instructions, so as not to overwhelm the intended user. To make the language powerful, we incorporate the new motion control and path tracking technique.

B. DESIGN GOALS

Originally, it was planned to incorporate all modules into the new language. Time dictated otherwise, due to a long procurement process in obtaining a new central processing unit and mother board. It was opted to design only the foundation, path tracking and motion control modules. This placed a constraint of compatibility, since implementation wanted to be realized in the interim and the other low level modules are written in assembly and C.

The overall design goal is to create a program structure that would stand the "test of time". Meaning, it should be portable and provide easy implementation of refinements and or advancements. With this in mind, a means to build abstractions and specialize them is the key. An object-oriented approach not only permits these, its designed to focus on them [2]. Thus, the language of C++ was chosen, since it allowed both an object oriented paradigm and

backward compatibility with the existing C library. Also, the existing code could be convert with very little effort. Other basic design goals are as follows:

- Maximize efficiency.
- Provide simple maintainability.
- Utilize structures that present maximum growth potential
- Maintain a small set of instruction.
- Allow flexibility with parameters.

C. THESIS ORGANIZATION

Chapter II of this thesis provides an overview of two techniques utilized for path planning and motion control. It begins with a basic overview of the current technique along with it's shortcomings and then affords a more in-depth look at the new technique for which OOPS-MML is predicated.

Subsequent chapters focus on the software design in response to the design goals mentioned above. Specifically, Chapter III contains the details of the abstract design and implementation of the basic data structures. Chapter IV discusses the building blocks utilized for path construction and Chapter V provides details with regards to interaction between these building blocks. Chapter VI outlines and presents the high level functionality of the new OOPS-MML language. Chapter VII contains details of the design and implementation of the architecture and its data structures. Chapter VIII summarizes the work and provides avenues for future work.

II. PATH TRACKING AND MOTION CONTROL RESEARCH

A. BACKGROUND

The scheme of path planning for Yamabico-11 is accomplished by a user writing a detailed program outlining a desired behavior. The user's program is interpreted into a path for the robot to track by generating intermediate reference postures that take into account desired velocity and locomotion capabilities. The robot then tracks through the path by requiring the odometer readings to mimic each intermediate posture before continuing to the next. A depiction of two specified postures and the intermediate postures generated can be seen in Figure 1.

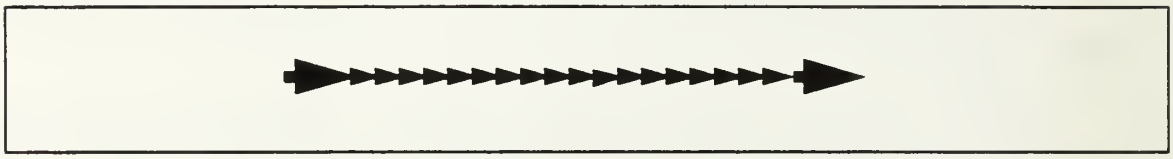

Figure 1: Path Tracking By Intermediate Reference Postures

Current research being conducted at NPS has proven shortcomings within this schema. The first is the requirement of "prior knowledge needed" of all transition points to precisely specify the postures needed when outlining the user's program. This is due to the lack of restrictions on the bounds of movement between two adjacent postures. This "prior knowledge needed" is only compounded as the level of complexity for traversal increases within an environment because there is a direct correlation between complexity and number of postures required to be specified. This problem can be visualized in Figure 2, a simple corridor with two obstacles requiring five pre-known specified postures. The second is due from research involving odometer correction and wall following[3]. Since the robot is pulled through the path, any odometer corrections result in jerky non-smooth motions, due to the robot's requirement to exactly mimic a posture on the Cartesian plane before continuing [4]. Thus, odometer correction placed behind the robot results in backward jerky motion. A correction placed in front results in accelerated performance, sometimes beyond operating parameters. Both corrections cause increased wheel slippage which only compounds odometer errors.

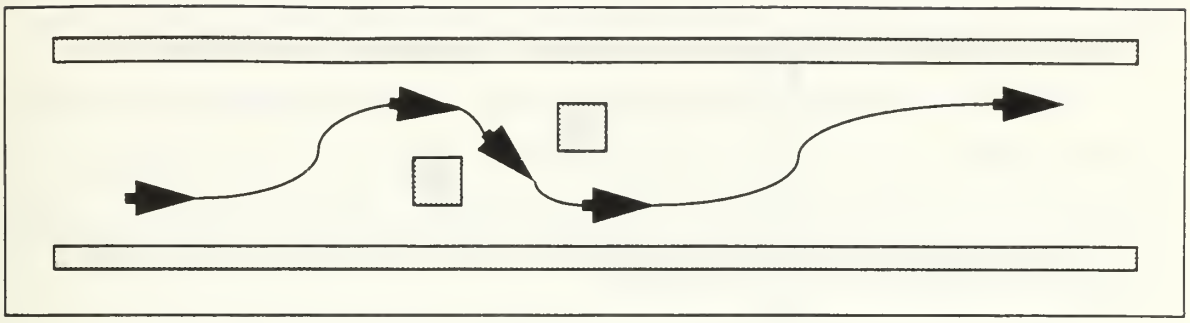

Figure 2: Pre-Known Postures Needed To Create A Path

B. PATH CONTROL BY CURVATURE

This technique far surpasses the subservient nature currently employed requiring the behavior of mimicking generated postures on a cartesian plane[5]. The new scheme greatly increases the robot's overall autonomous growth by allowing the robot to take on reactive behaviors.

The power of this scheme comes from the concept of controlling robot motion solely through the manipulation of kappa κ or the curvature value. This is implemented by adding the kappa value to the existing posture structure. The elementary structure of this scheme is defined as a configuration and is represented by a quadruple (x, y, θ, κ). This representation not only affords the means to accurately describe the robot's positioning, it also allows a way to represent a variety of geometrical path forms. These path forms are then utilized as references to calculate the desired kappa for the robot to follow and if required converge onto that path form.

Path planning is still accomplished by a user writing a detailed program. Although instead of specifying postures, a series of templates called path elements are specified. Within this series, a relationship joins each path element with it's successor and forms one continual path. The path elements are capable of taking on the shape of a variety of path forms. Path forms are geometrical representations that can be mathematical defined as a configuration, in part or combination thereof. Originally this schema was predicated upon path forms of a line

and a circle [1]. Currently this work is being refined and expounded upon to included a path form of a parabola [5]. With hopes of including cubic spiral in the near future.

To reveal the intricacies of this technique Figure 3 depicts a scenario. Here we have a path specified as a series of two infinite path elements. The path elements p1 and p2 are in the path forms of a line, the relationship between them being an intersection. The robot is currently utilizing the path form of p1 as a reference.

Reactive behavior comes about by allowing the robot from any configuration to project itself perpendicularly onto p1. This projection currently being done every 10 msec produces an image on the path form. This image is the configuration the robot would be if it was exactly on the path form. Also it provides the position that is the minium distance between the robot and the path form. The robot then calculates a new kappa that will follow p1 and reduce the distance between the robot and it's image.

The size constant s0 is a user defined variable utilized to determine the sharpness of curvature for which to converge onto a path form. This is used in two ways. The first is with the calculation of a new kappa. The s0 value is factored into the equation to determine a kappa to reflect the anticipated sharpness. The second is the transition point between the relationship of two path elements. This is approached from a backward viewpoint. In that given the value of s0, at which point must the robot start projecting it's image onto p2 in order to not overshoot.

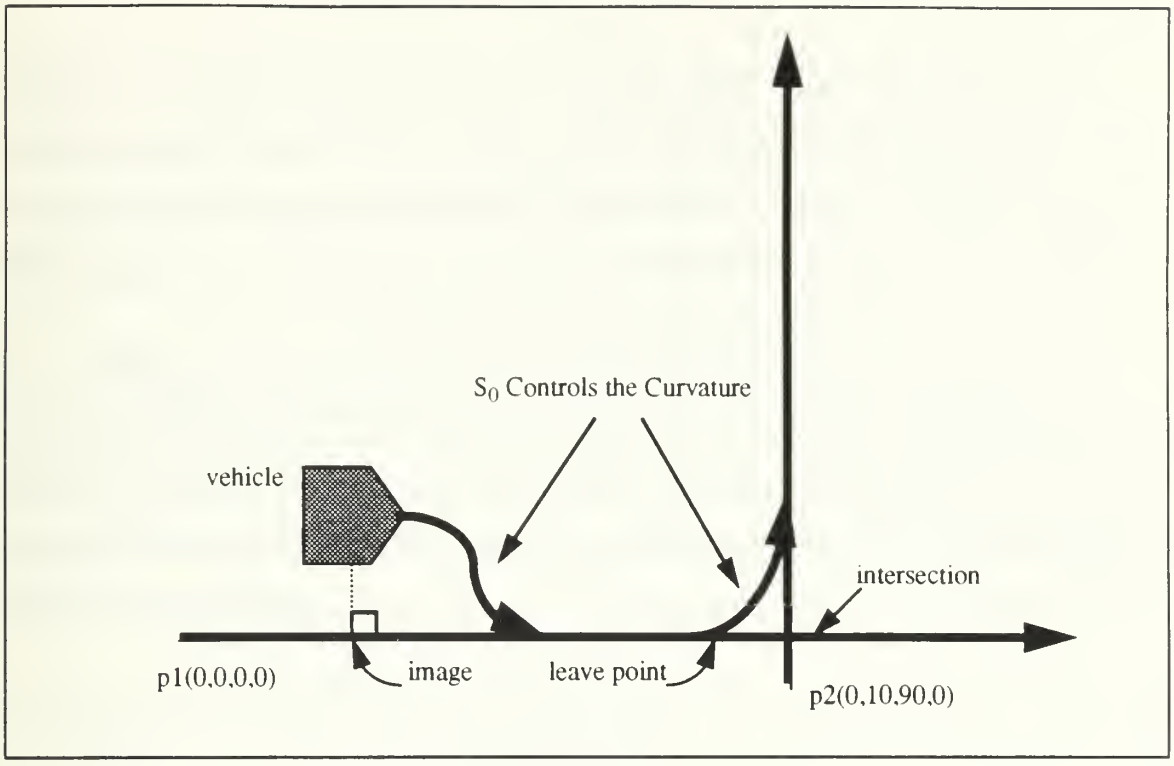

Figure 3: Path Tracking

III. BASIC STRUCTURE DESIGN

The basis of the OOPS-MML design is to be naturally expressive and provide tight coupling between design and implementation in order to support the design goals addressed. To accommodate this, it is paramount that relationships in the structure of implementation be captured. The structure of implementation for this new technique lies within the different graphical path forms. It is these objects that must be captured.

The premiss behind this new technique of path control by curvature is all path forms can be represented with the use of the elementary structures of a configuration. Although to facilitate all design goals, the need for an application independent way to abstractly describe, manipulate and manage the different path forms as individual entities is paramount. This can be accomplished by incorporating the domain in a public inheritance schema, see Figure 4.

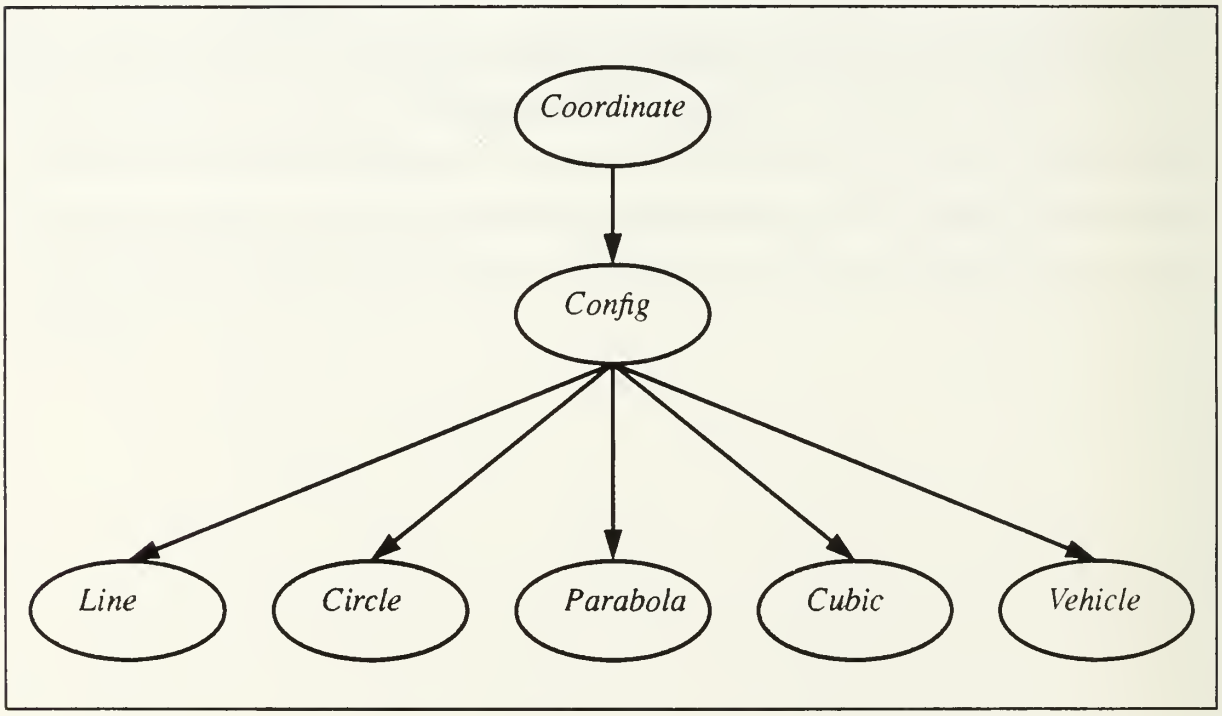

Figure 4: Inheritance Schema

This representation satisfies the overall basis of design for OOPS-MML. It abstractly describes the individual entities while providing the use of virtual functions as a means of manipulation and management. Also, it directly supports simple maintenance, growth potential and efficiency by incorporating levels of abstraction and encapsulation. The remainder of this chapter elaborates the syntax of the constructors and a component description of each object class. Actual code can be seen in Appendix A.

A. BASIC OBJECT CLASS STRUCTURES SYNOPSIS

The following are data classes of the unshaded objects depicted in Figure 4. These are low level structures within the inheritance schema and are typically hidden from the user. Although they can be utilized as utilities. The constructors automatically calculate and set all values within that object.

1. Coordinate Object Class (Coordinate)

Syntax: Coordinate(double x, double y)

Coordinate(Coordinate &c)

Description: This class is designed to represent a coordinate on a two dimensional cartesian plane, see Figure 5. The components are *_X* and *_Y*. The *_X* value is utilized to describe the x position. The *_Y* value is utilized to describe the y position.

```
class Coordinate
  {
    public:

    // Structure
       double _X;
       double _Y;
  }
```

Figure 5: C++ Code Example Of Coordinate Class Structure

2. Configuration Object Class (Config)

Syntax: Config(double x, double y, double t)

Config(double x, double y, double t, double k)

Config(Coordinate &p, double t, double k)

Config(Config &q)

Description: This class is designed to represent the elementary object of the quadruple utilized in path planning by curvature, see Figure 6. The inherited component is a *Coordinate*. The additional components are *_Theta* and *_Kappa*. The *Coordinate* value is used to describe the x and y position. The *_Theta* value is used to describe the orientation. The *_Kappa* value is utilized to describe the curvature.

```
class Config : public Coordinate
  {
    public :

    // Structure
       double _Theta;
       double _Kappa;
  }
```

Figure 6: C++ Code Example Of Config Class Structure

3. Vehicle Object Class (Vehicle)

Syntax: Vehicle(double x, double y, double t, double k)

Vehicle(double x, double y, double t, double k, double s)

Vehicle(Config &q)

Vehicle(Vehicle &v)

Description: This class is designed to represent the vehicle odometer settings, see Figure 7. The inherited component is a *Config*. The additional components are *_Speed*, *_Omega*, *_S0*, *_L_Accel* and *_R_Accel*. The *_Speed* value represents the desired speed for the vehicle and can be set

by the user or the default value of 30 cm/sec can be used. The *_Omega* value represents the omega of the vehicle and set by multiplying the kappa and speed value of the structure. The *_S0* value represents the s0 variable utilized to calculate a new kappa value for the vehicle. This can be either set by the user or the default value of 15 cm can be used. The *_L_Accel* value represents the linear acceleration for the vehicle and is set by the user or the default value of 20 cm/sec can be used. The *_R_Accel* value represents the rotational acceleration for the vehicle and is either set by the user or the default value of 10 cm/sec can be used.

```
class Vehicle : public Config
  {
    public :

    // Structure
       double _Speed;
       double _Omega;
       double _S0;
       double _L_Accel;
       double _R_Accel;

  }
```

Figure 7: C++ Code Example Of Vehicle Class Structure

B. PATH FORM OBJECT CLASS SYNOPSIS

The following are data structures of the shaded objects depicted in figure 3. These are the high level structures in the inheritance schema and are visible to the user. They represent the desired path form. The constructors automatically calculate and set all values within that structure.

1. Line Object Class(Line)

Syntax: Line(double x, double y, double t)

Line(double x, double y, double t, double k)

Line(Coordinate &p1, Coordinate &p2)

Line(Config &q)

Line(Line &l)

Description: This class is designed to represent the path form of an infinite line or a vector, see Figure 8. The inherited component is a *Config*. This *Config* represents the elementary path form of an infinite line by having a *_Kappa* value of zero. The additional components are a *_P2*, *_A*, *_B* and *_C*. The *_P2* represents a projected coordinate on the line represented by the *Config* values. This is used to allow the line to have a second form of Ax + By + C = 0. The *_A*, *_B* and *_C* represent the constants of this equation.

```
class Line : public Config
  {
    public:

    // Structure
       Coordinate _P2;
       double _A;
       double _B;
       double _C;
  }
```

Figure 8: C++ Code Example Of Line Class Structure

2. Circle Object Class(Circle)

Syntax: Circle(double x, double y, double t, double k)

Circle(Config &q)

Circle(Circle &c)

Circle(double cx, double cy, double r)

Circle(Coordinate c, double r)

Description: This class is designed to represent the path form of a circle, see Figure 9. The inherited component is a *Config*. The *Config* represents the elementary path form of a point on the path form of a circle. The

additional components are *_Center* and *_Radius*. The *_Center* represents the circle's center coordinate. The *_Radius* represents the circle's radius. The sign of the radius determines rotational direction. Positive for counter clockwise and negative for clockwise. If constructed by center and radius the *Config* is calculated from 90 degrees.

```
class Circle : public Config
  {
    public:

    // Structure
       Coordinate  _Center;
       double      _Radius;
  }
```

Figure 9: C++ Code Example Of Circle Class Structure

3. Parabola Object Class (Parabola)

Syntax: Parabola(double dx, double dy, double dt, double fx, double fy)

Parabola(Config &q, Coordinate &c)

Parabola(Line &l, Coordinate &c)

Parabola(Parabola &p)

Description: This class is designed to represent the path form of a parabola, see Figure 10. The inherited component is a *Config*. The *Config* represents the parabola's directrix. The additional component is the *_Focus*. The *_Focus* represents the parabola's focus coordinate.

```
class Parabola : public Config
  {
    protected:

    // Structure
       Coordinate _Focus;
  }
```

Figure 10: C++ Code Example Of Parabola Class Structure

4. Cubic Spiral Object Class (Cubic)

Syntax: Cubic(double x, double y, double t)

Cubic(Coordinate &c, double t)

Cubic(Config &q)

Cubic(Cubic &c)

Description: This class is designed to represent the path form of a cubic spiral, see Figure 11. The inherited component is a *Config*. The *Config* represents the cubic spiral's posture with *_Kappa* value of zero. The additional components are *A_value* and *L_value*. The *A_value* is the area of the cubic spiral curve. The *L_value* is the length of the cubic spiral.

```
class Cubic : public Config
  {
    protected:

    // Structure
       double A_value;
       double L_value;
  } ;
```

Figure 11: C++ Code Example Of Cubic Spiral Class Structure

IV. PATH ELEMENTS

The basis for path planning for the new technique of path control by curvature is to provide the means to specially construct a path form for a particular use. This is accomplished by designing templates that can take a desired path form as a parameter and bind them to a specific behavior. These templates are called path elements and are the building blocks utilized to construct a continual path for the robot to follow.

To accommodate these path elements in OOPS-MML, it requires only four templates because the C++ paradigm allows overloading of functions. This affords a powerful yet simplistic means to express robotic motion. Also, this directly supports the design goals of providing a small set of clear concise instructions and allowing flexibility with parameters.

A. A PATH (Path)

A Path can be geometrically described in the path forms of a *Line*, *Circle* or *Parabola* as illustrated in Figure 12. This path element is designed to allow the robot to follow a path form indefinitely with no constraints placed upon it's starting or ending points. Therefore, the path form values are used only to describe the path form itself.

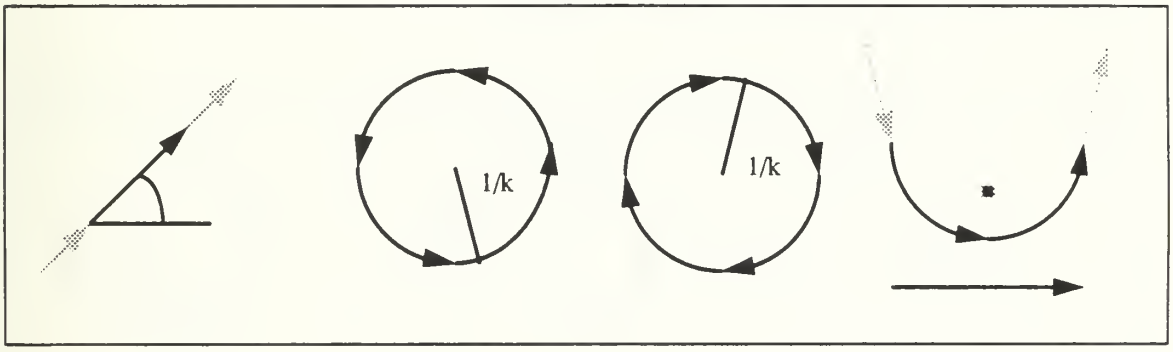

Figure 12: Examples Of Path Element Forms

B. A FORWARD PATH (FPath)

A FPath can be geometrically described in the path forms of a *Line* or *Circle* as illustrated in Figure 13. This path element is designed to allow the robot to follow a path form indefinitely

with a constraint placed upon it's starting point. Therefore, the path form values are used for two purposes. The first is to describe the path form itself. The second is to describe the starting point the robot must enter to follow the path.

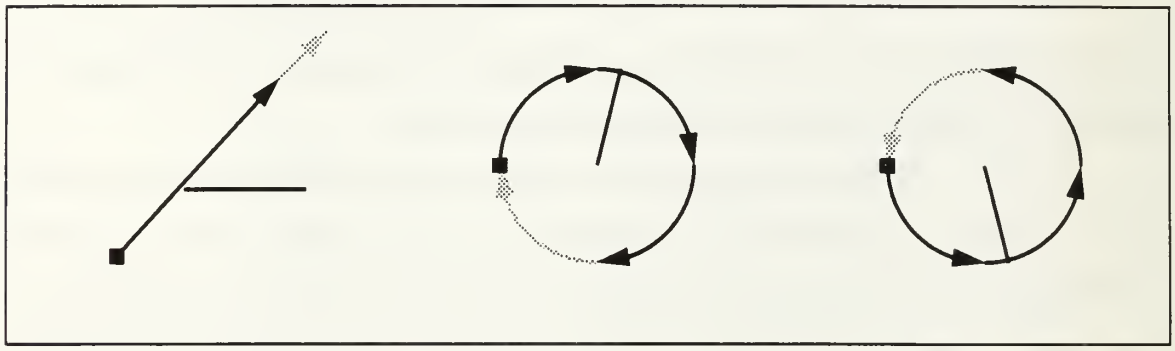

Figure 13: Examples Of Forward Path Element Forms

C. A BACKWARD PATH (BPath)

A BPath can be geometrically described in the path forms of a *Line* or *Circle* as illustrated in Figure 14. This path element is designed to allow the robot to follow a path form with a constraint placed upon it's ending or leaving point. Therefore, the path form values are used for two purposes. The first is to describe the path form itself. The second is to describe the point the robot must terminate following the path.

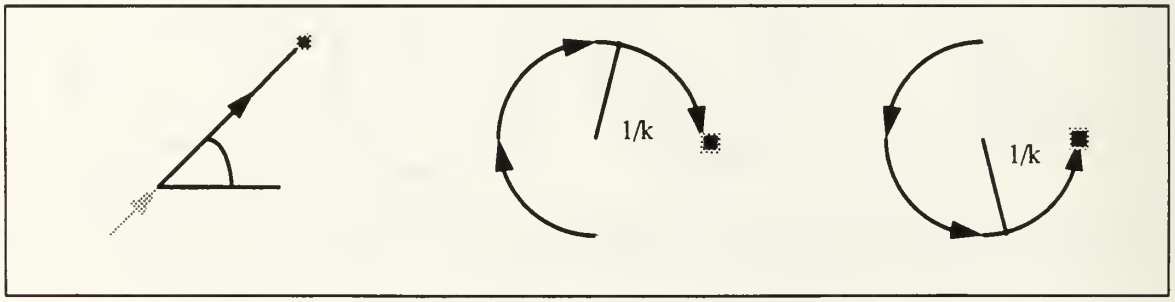

Figure 14: Examples Of Backward Path Element Forms

D. A POSTURE (Posture)

A Posture can be geometrically described in the path form of a *Cubic*. This is designed to allow the robot to follow a cubic spiral with a specific starting and ending points. In order for this to be a complete path element, it must be used in conjunction with another posture, in front of a FPath or after a BPath as illustrated in Figure 15.

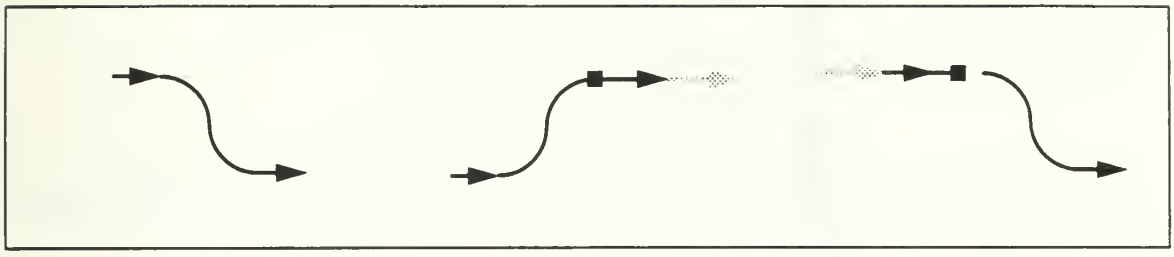

Figure 15: Examples Of Posture Combinations

V. PATH CONSTRUCTION

Chapter IV introduced the foundation for Yamabico's path planning by defining four path elements. In this chapter, we develop the concept to construct a path utilizing these path elements. Figure 16 illustrates this concept with a path made up of three path elements in the path forms of a Line, Circle and Line. The overall objective is to provide a smooth uniform path for a rigid body robot to track. To accomplish this we must establish the basis and modes of interaction between the path elements.

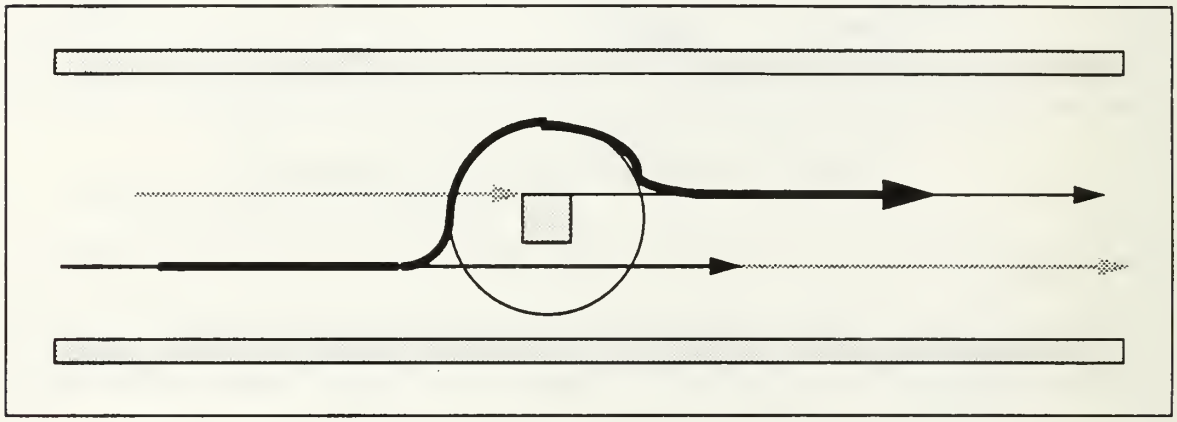

Figure 16: A Path Build By Three Path Elements

A. THE FOUNDATION

The basis for constructing a path from path elements is to allow continuity of progression. Formulating an order of progression is accomplished by representing a path as a sequence of one or more path elements. Continuity is achieved by recursively linking each path element within the sequence to it's successor. In order to link consecutive path elements, we must establish if a relationship exists between them and if so what it is.

Table 1 illustrates the possible combinations of consecutive path elements. The shaded blocks represent non-permissible combinations. Utilization of these results in an error and termination of the program. The unshaded blocks represent permissible combinations and in each block is an entry that defines their relationship. The relationships are the translatory

Table 1: PERMISSIBLE COMBINATIONS AND TRANSLATORY MODES

From To	Path(Line)	Path(Circle)	Path(Parabola)	FPath(Line)	FPath(Circle)	BPath(Line)	BPath(Circle)	Posture(Cubic)
Path(Line)	TRM	TRM	TRM			TRM	TRM	
Path(Circle)	TRM	TRM				TRM	TRM	
Path(Parabola)	TRM					TRM	TRM	
FPath(Line)	TRM	TRM	TRM			TRM	TRM	
FPath(Circle)	TRM	TRM	TRM			TRM	TRM	
BPath(Line)	TREM	TREM	TREM	CSM	CSM	TREM	TREM	CSM
BPath(Circle)	TREM	TREM	TREM	CSM	CSM	TREM	TREM	CSM
Posture(Cubic)				CSM	CSM			CSM

modes from the path to it's successor. There are three types of translatory modes and each is elaborated in the rest of this chapter.

B. TRANSLATORY MODES

1. Transition Required Mode (TRM)

This mode is utilized when the path element is a Path and it's successor is a BPath or another Path. With this transition mode, two methods for behavior are possible. The first is classified as an unbounded transition. Once the robot terminates the current path, it will simply converge onto it's successor path. This behavior is dependent upon the robot's proximity to the successor and it's current value for s0. Unbounded transitions are caused by the path elements having no common intersect coordinates. Three examples of two parallel Path elements in Line object form and their behavior are shown in Figure 17. For each, the robot terminates p1 immediately upon addition of p2 to the sequence.

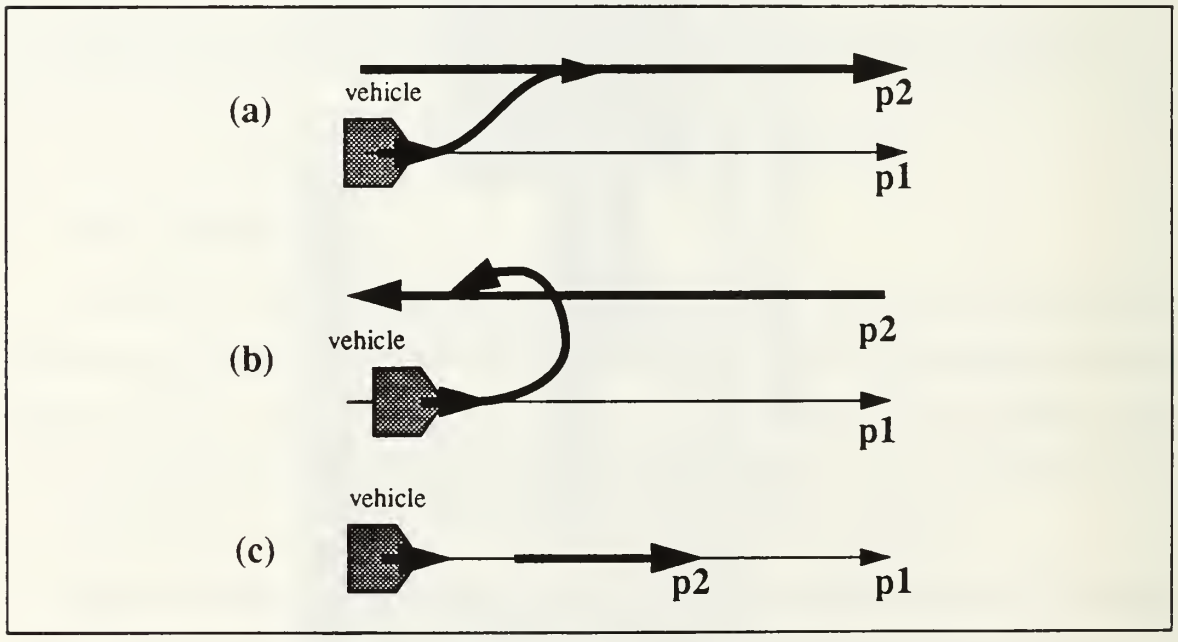

Figure 17: Examples Of Unbounded Behavior Between Parallel Line Object Paths

The second is classified as a bounded transition. Bounded transition is utilized when a path element and it's successor have common intersect coordinates. This transition is formulated by the path projection method [5]. The method with the current value for s0 projects simulated tracks to determine when to leave a path element in order to achieve an optimum transition path. An optimum transition path is one that leaves the current path element at the shortest distance from the intersection and does not cross the successor's path. The distance is formulated by binary gates of multiples of s0. Figure 17 depicts four projected tracks resulting in t4 as the optimum track with a leaving distance of 1.75s0.

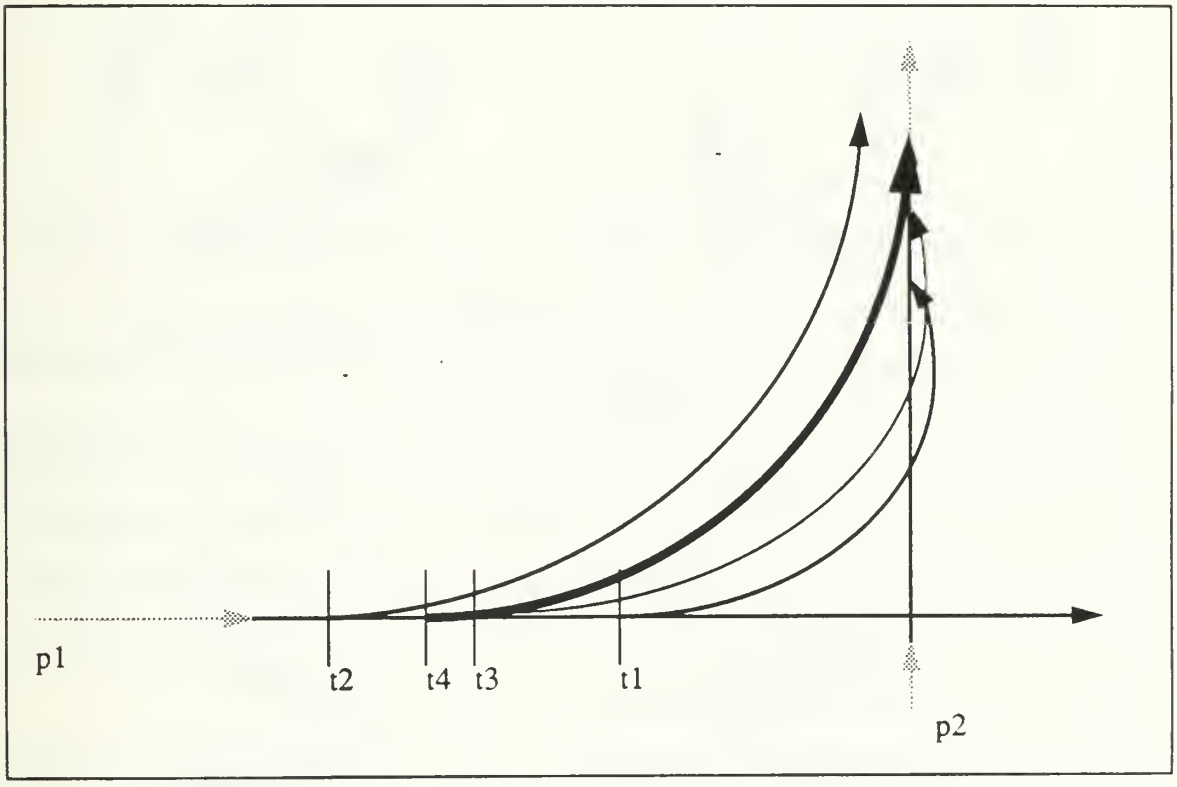

Figure 18: Transition Projections Between Two Parallel Straight Lines

Although these are the general concepts for formulating both transitions, their behavior may be slightly modified dependant on the path forms. Therefore it is necessary to show the idiosyncrasies of each combination of path forms.

a. Line and Circle Objects

(1) Unbounded Behavior. When a *Line* object and *Circle* object are non-intersecting, a termination point on the current path element is calculated. This point is the closest distance from the *Line* object to the center of the *Circle* object. Figure 19 illustrates the behavior when the *Line* and *Circle* objects have the same directionality. Figure 20 illustrates the behavior when the *Line* and *Circle* objects have different directionality.

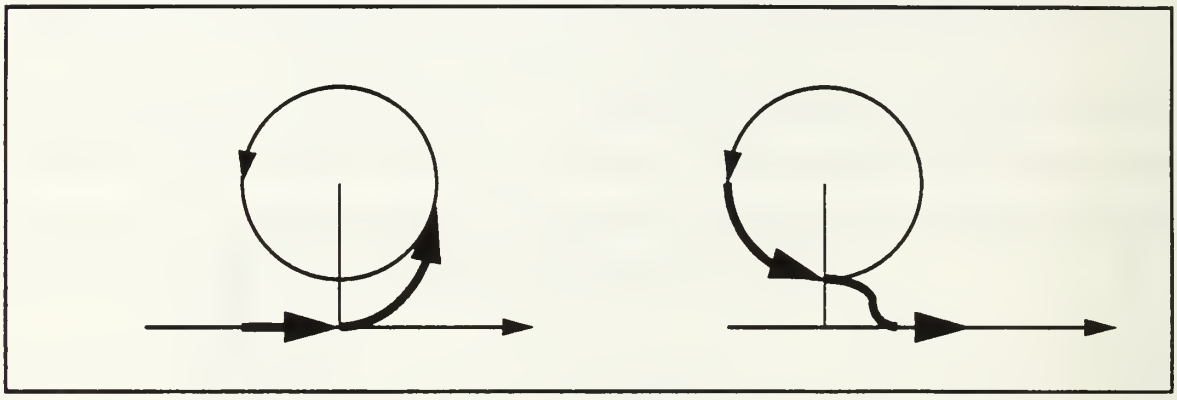

Figure 19: Examples Of Unbounded Behavior Between Line And Circle Objects Having Similar Directionality

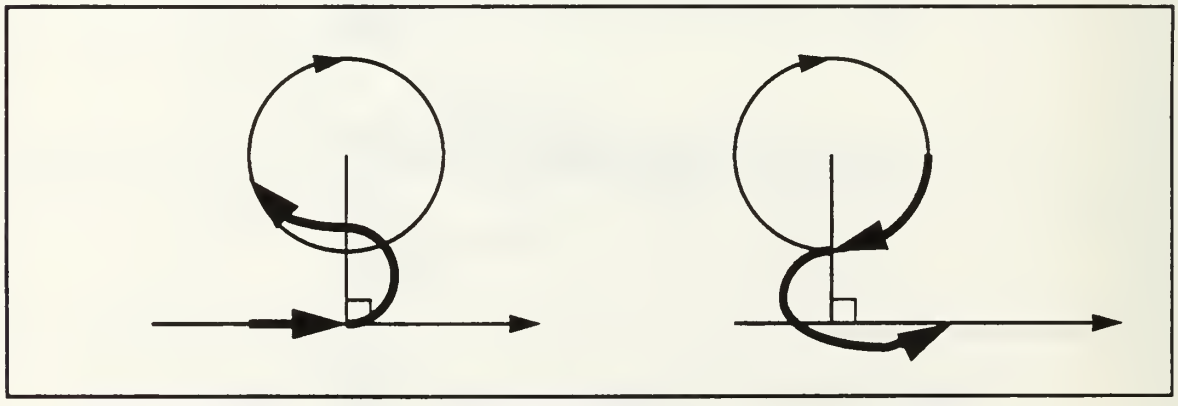

Figure 20: Examples Of Unbounded Behavior Between Line And Circle Objects Having Different Directionality

(2) Bounded Behavior. When a *Line* Object and a *Circle* object are intersected, there are normally two intersect coordinates to consider. They are defined as upwind and downwind intersect points. Thus, the transition point is dependant on the position of the robot. If the robot is outside of the circle then the upwind intersection point is used in the path projection method, see Figure 21 (a). If the robot is inside of the circle, then the downwind intersection point is used in the path projection method, see Figure 21 (b).

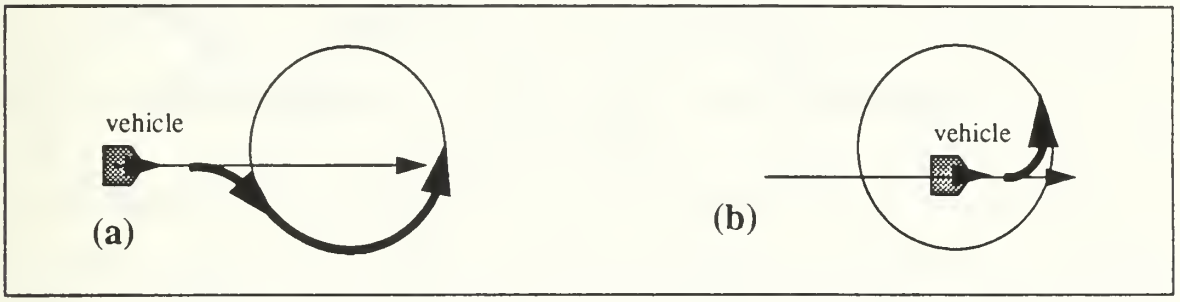

Figure 21: Examples Of Bounded Behavior Between Line And Circle Objects

b. Line and Parabola Transitions

(1) Unbounded Behavior. When a *Line* and *Parabola* objects that are non-intersecting, a termination point on the current path element is calculated. Although, this point is the closest distance from the *Line* object to the focus of the *Parabola* object. Figure 22 illustrates the behavior when *Line* and *Parabola* objects have the same directionality. Figure 23 illustrates the behavior when the *Line* and *Parabola* objects have different directionality.

(2) Bounded Behavior. When a *Line* Object and a *Parabola* object are intersected, transitions are the same as for *Line* and *Circle* objects. If the robot is outside of the *Parabola* object then the upwind intersection point is used in the path projection method, see Figure 24 (a). If the robot is inside of the parabola then the downwind intersection point is used in the path projection method, see Figure 24(b).

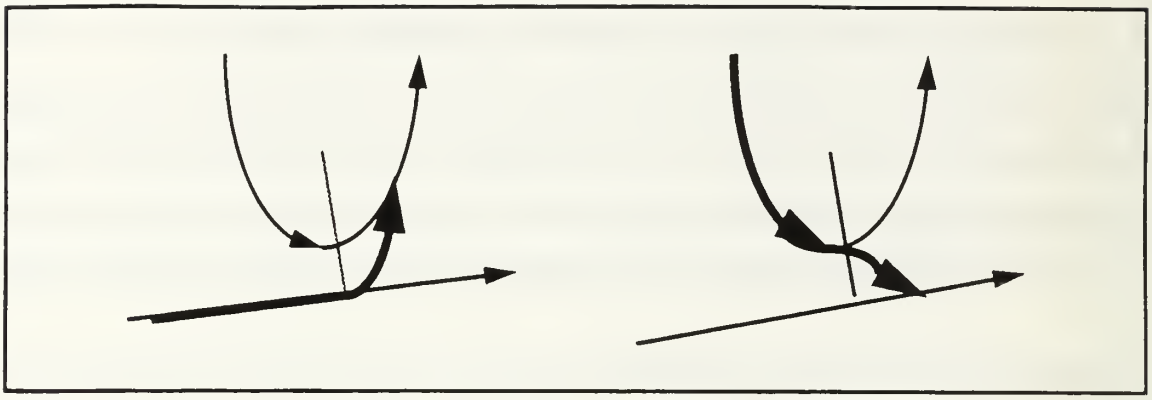

Figure 22: Examples Of Unbounded Behavior Between Line And Parabola Objects Having Same Directionality

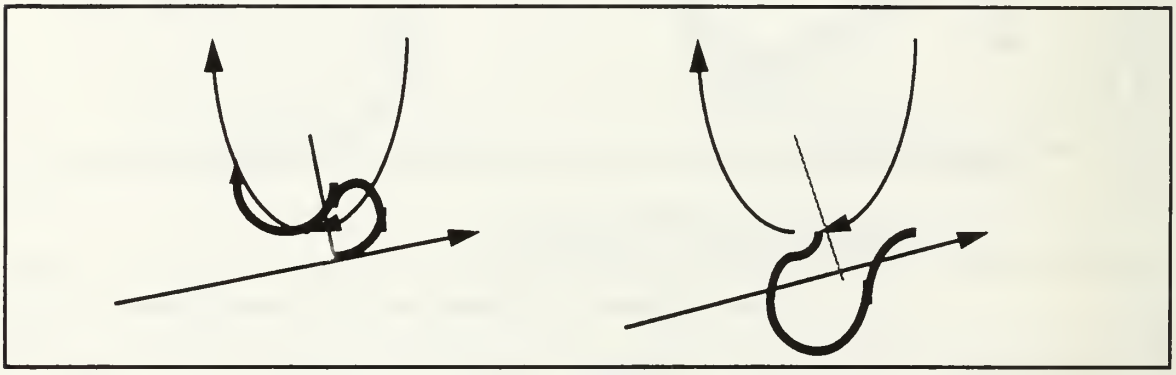

Figure 23: Examples Of Unbounded Behavior Between Line And Parabola Objects Having Different Directionality

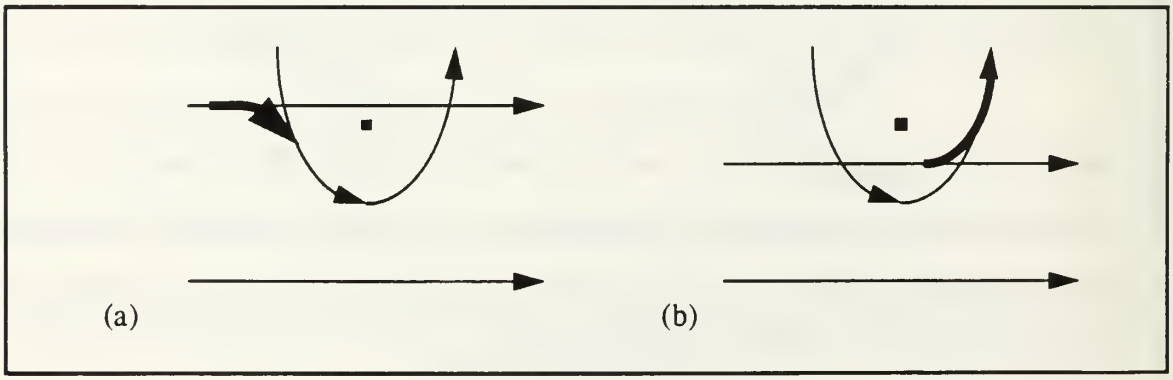

Figure 24: Examples Of Bounded Behavior Between Line And Parabola Objects

c. *Circle and Circle Transitions*

(1) Unbounded Behavior. When *Circle* and *Circle* objects are non-intersecting, a *Line* object is inserted between their center coordinates. This converts the behavior to a bound behavior. Figure 25 illustrates this insertion of a *Line* object between the *Circle* objects.

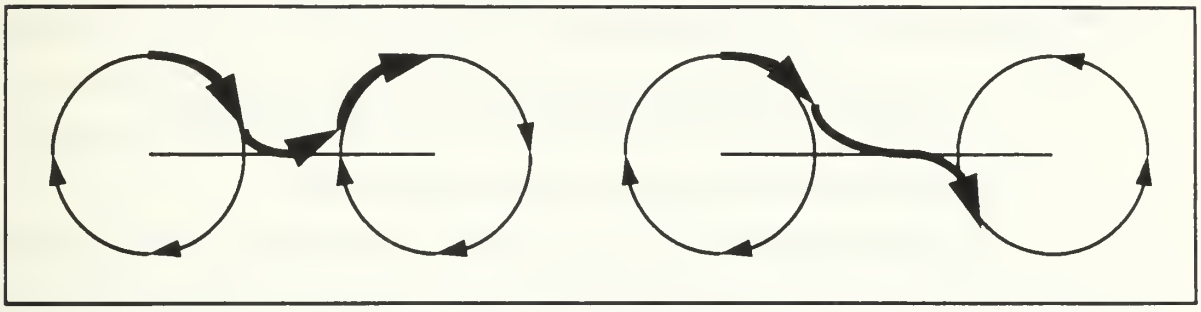

Figure 25: Examples Of Non-Intersecting Circle Objects And Insertion Of A Line Object

(2) Bounded Behavior. When *Circle* and *Circle* objects intersect, transitions are the same as for *Line* and *Circle* objects. The upwind intersection point is used in the path projection method, see Figure 26.

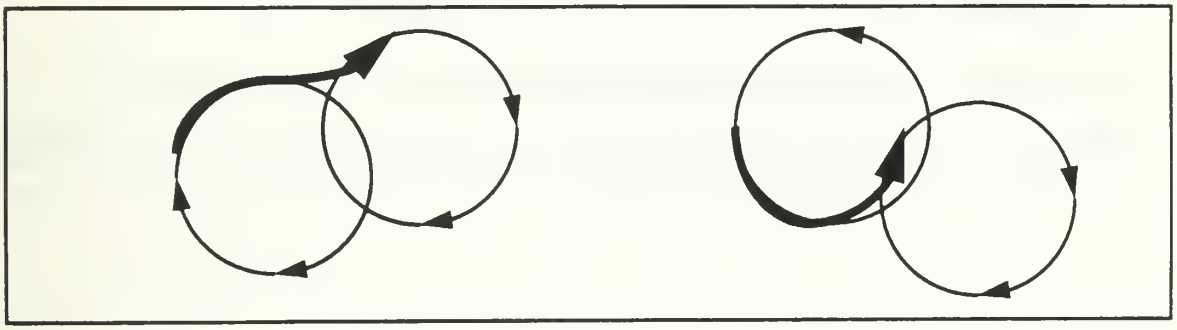

Figure 26: Examples Of Intersecting Circle Objects

2. Transition Required at Endpoint Mode (TREM)

This mode is utilized when the robot's path element is a BPath and it's successor is a Path or another BPath. In this mode, the behavior is similar to the unbounded behavior of

Line objects. The termination point for the robot to project it's image onto the successor is predetermined by the user, see Figure 27.

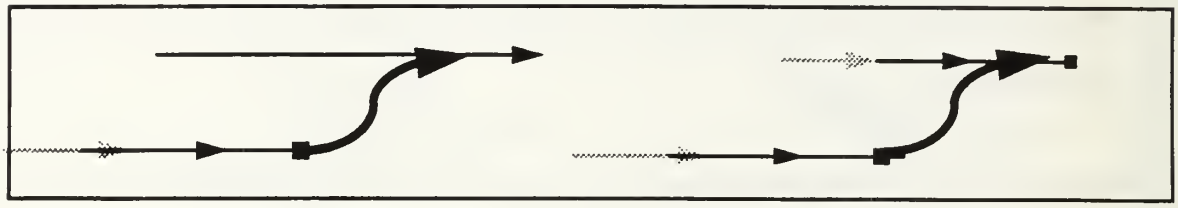

Figure 27: Examples Of BPath Line Object And Line Objects

3. Transition Required by Cubic Spiral Mode (CSM)

This mode is utilized when the path element is either a BPath or a Posture and their successor is a FPath or a Posture. In this case, a cubic spiral curve is created and inserted between the path elements.

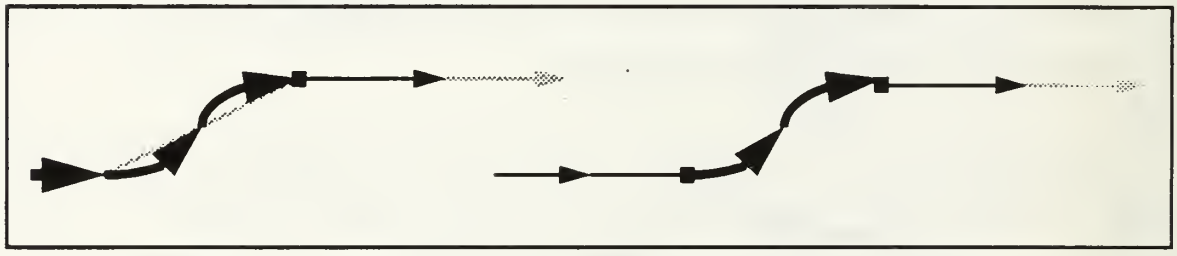

Figure 28: Examples Of Cubic Spirals For A Posture To FPath And A BPath To FPath

VI. OOPS-MML COMMANDS

The OOPS-MML commands are function calls utilized by a user for creating a program to outline and control a robot's behavior. These commands are logically broken down into three categories, as seen in Figure 29. Parameter commands include function calls that establish characteristics for behaviors and provide communications between the user's program and the robot. Stationary commands include function calls utilized for stopping robot motion or that have being motionlessness as a prerequisite. The motion commands include functions calls used for robot motion and path planning. Actual code for each of these is in Appendix B.

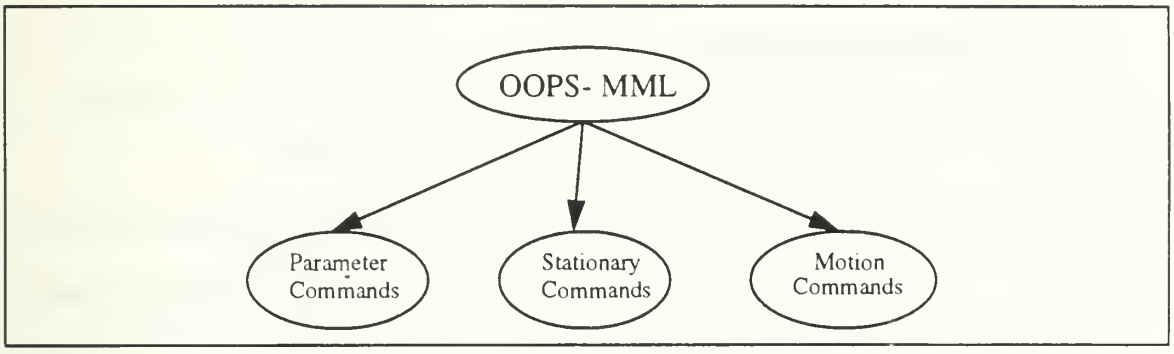

Figure 29: OOPS-MML's Command Function Schema

Within these categories, there are certain functions that are similar. These functions are designed to allow a desired command to either take effect sequentially or immediately. Sequential functions are loaded into a buffer and take effect in the same order they appeared in the user's program. Immediate functions take effect as soon as they are called. Normally immediate functions are utilized in conjunction with a conditional statement or branch in the user's program.

A. THE PARAMETER COMMANDS

1. Set Size Constant (Set_S0)

Syntax: void Set_S0(double s)

Type: Sequential

Description: This function is utilized to set the size constant s0 for the tracking algorithm. This parameter defines an intended sharpness for the curvature. The greater the s0 the more intense or sharper the curvature. The default value for s0 is 15 cm.

2. Reset Size Constant (Reset_S0)

Syntax: void Reset_S0(double s)

Type: Immediate

Description: This function is utilized to reset the size constant s0 for the tracking algorithm on the fly.

3. Set Robot Speed (Speed)

Syntax: void Speed(double s)

Type: Sequential

Description: This function is utilized to set the intended speed for the robot. The robot will smoothly accelerate to this speed utilizing the value for acceleration. The default value is 30 cm/sec and the maximum allowed is 60 cm/sec.

4. Reset Robot Speed (Reset_Speed)

Syntax: void Reset_Speed(double s)

Type: Immediate

Description: This function is utilized to reset the robot's speed on the fly.

5. Set Robot Acceleration (Set_Acc)

Syntax: void Set_Acc()

Type: Sequential

Description: This function is utilized to set the robot's acceleration. The acceleration is utilized for the rate of increase or decrease to achieve the desired speed. The default value is 20 cm/sec squared.

6. Reset Robot Acceleration (Reset_Acc)

Syntax: void Reset_Acc()

Type: Immediate

Description: This function is utilized to reset the robot's acceleration on the fly.

7. Set Robot Configuration (Set_Rob)

Syntax: void Set_Rob(double x, double y, double t)

void Set_Rob(double x, double y, double t, double k)

void Set_Rob(double x, double y, double t, double k, double s)

void Set_Rob(Config &q)

void Set_Rob(Vehicle &q)

Type: Sequential

Description: This function is utilized to set the odometer when the robot's mode is stationary. It sets all components of the robot odometer structure to the referenced structure's compatible components. If a *Config* structure is passed the robot's speed and omega values are set to zero. Normally used at the start of the OOPS-MML user program, it can be utilized throughout given the robot's mode condition is satisfied

8. Set Robot Configuration (Reset_Rob)

Syntax: void Reset_Rob(double x, double y, double t)

void Reset_Rob(Config &q)

void Reset_Rob(Config &q)

void Reset_Rob(Vehicle &q)

Type: Immediate

Description: This function is utilized to set the odometer on when the robot's mode is either stationary or moving. It sets the _X, _Y and _Theta components of the robot odometer structure to the referenced structure's compatible components. It is normally used to instantaneously correct robot

odometery errors. The kappa or omega of the robot should not be reset since both are calculated in relation to remaining or converging onto the current path element and an instantaneous change would not take into account the current path. In Figure 30, the robot is tracking with an odometry error. In this case, Reset_Rob is used to reset the robot odometer to the actual or correct position.

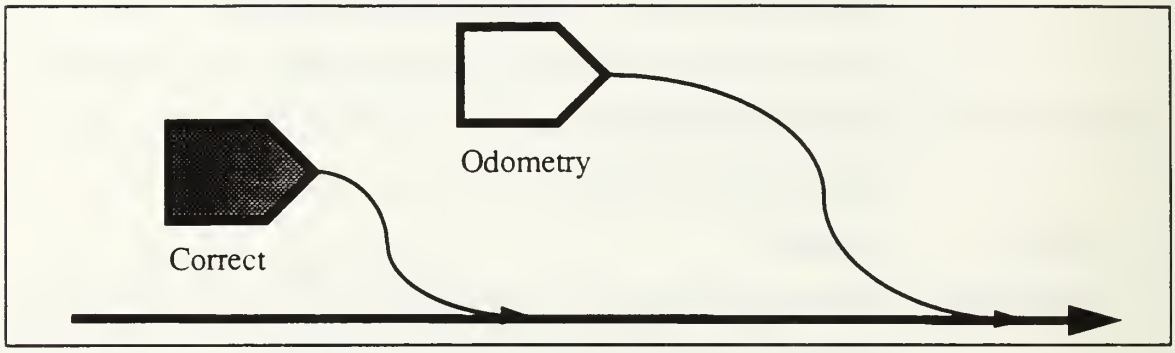

Figure 30: Reset_Rob Function

9. Get Robot Configuration (Get_Rob)

Syntax: void Get_Rob(Config &q)

void Get_Rob(Vehicle &q)

Type: Immediate

Description: This function is utilized when the robot's mode is either stationary or moving. It retrieves the current robot odometer settings. This is accomplished by replacing the compatible components of the structure sent with the robot odometer settings

10. Get Current Buffer Object (Get_Buf)

Syntax: void Get_Buf(List &q)

Type: Immediate

Description: This function is utilized to obtain the current record object from the buffer. This is accomplished by replacing the compatible components of the class object sent with the current record object of the buffer.

11. Trace Robot (Trace_Robot)

Syntax: void Trace_Robot()

Type: Sequential

Description: This function is utilized to record the robot's movements to a file. The file contains the robot's odometer coordinates as it tracks the path. This can be utilized to scrutinize the data or to provide a means to represent the path two dimensionality with GNUPLOT.

12. Trace Simulator (Trace_Sim)

Syntax: void Trace_Sim()

Type: Sequential

Description: This function is utilized to record the robot's movements to a file. The file contains the robot's odometer settings and sonar returns as it tracks the path. This can be utilized to scrutinize the data or to provide a means to represent the robot's behavior in a two or three dimensional simulator.

13. Enable Sonar Group (Enable_Sonar)

Syntax: void Enable_Sonar(int g)

Type: Sequential

Description: This function is utilized to enable sonar group g. Once enabled, the group will become active and provide distance information from each specific sonar return.

14. Disable a Sonar (Disable_Sonar)

Syntax: void Disable_Sonar(int g)

Type: Sequential

Description: This function is utilized to disable sonar group g. Once disabled the group will become inactive and stop providing distance information.

B. STATIONARY COMMANDS

1. Stop Robot (Stop)

Syntax: void Stop()

Type: Immediate

Description: This function is utilized to stop the robot and to clear the existing buffer. Once this command is received the robot will decelerate to a smooth stop. The robot will then remain motionless and wait for a new command to be activated.

2. Stop Robot at a Specific Configuration (Stop)

Syntax: void Stop(double x, double y, double t)

void Stop(Config &q)

void Stop(Cubic &c)

Type: Sequential

Description: This Function is utilized to stop the robot at a particular place. Once this command is activated the robot will construct a path from a cubic spiral and stop by smoothly decelerating at the desired location. The robot will then remain motionless and wait for a new command to be activated.

3. Terminate Program (End)

Syntax: void End()

Type: Sequential

Description: This function is utilized to terminate the user's program. Once this command is activated the robot will decelerate to smooth stop and terminate the user's program.

4. Halt Robot Motion (Halt)

Syntax: void Halt()

Type: immediate

Description: This function is utilized to suspend robot motion. Once this command is received the robot will decelerate in smooth manner and remain motionless until the resume command is activated.

5. Resume Robot Motion (Resume)

Syntax: void Resume()

Type: immediate

Description: This function is utilized to resume robot motion. This command is used only after the command to halt the robot is activated. Once this command is received the robot will resume in the same manner as before it encounter the halt command.

6. Rotate Number of Degrees (Rotate)

Syntax: void Rotate(double d)

Type: Sequential

Description: This function is utilized to rotate the robot a specific number of degrees. The robot's current orientation will only increase or decrease dependent on the sign of the desired change.

7. Rotate to Theta (Rotate_To)

Syntax: void Rotate_To(double d)

Type: Sequential

Description: This function is utilized to rotate the robot to the desired orientation. The rotational directions will rotate in the most efficient way required to obtain the desired orientation.

C. MOTION COMMANDS

1. Move While Tracking a Path (Path).

Syntax: void Path(double x, double y, double t)

void Path(double x, double y, double t, double k)

void Path(Config &q)

void Path(Line &q)

void Path(Circle &q)

void Path(Parabola)

void Path(Line &, Coordinate&)

void Path(double dx, double dy, double dt, double fx, double fy)

Type: Sequential

Description: This function is utilized to accommodate the motion control and behavior of the path element defined as a Path. The robot will follow the appropriate path form desired from the specified parameters.

2. Move While Tracking a Forward Path (FPath)

Syntax: void FPath(double x, double y, double t)

void FPath(double x, double y, double t, double k)

void FPath(Config &q)

void FPath(Line &q)

void FPath(Circle &q)

Type: Sequential

Description: This function is utilized to accommodate the motion control and behavior of the path element defined as a FPath. The robot will generate a cubic spiral path form to enter and follow the appropriate path form desired from the specified parameters, see Figure 31.

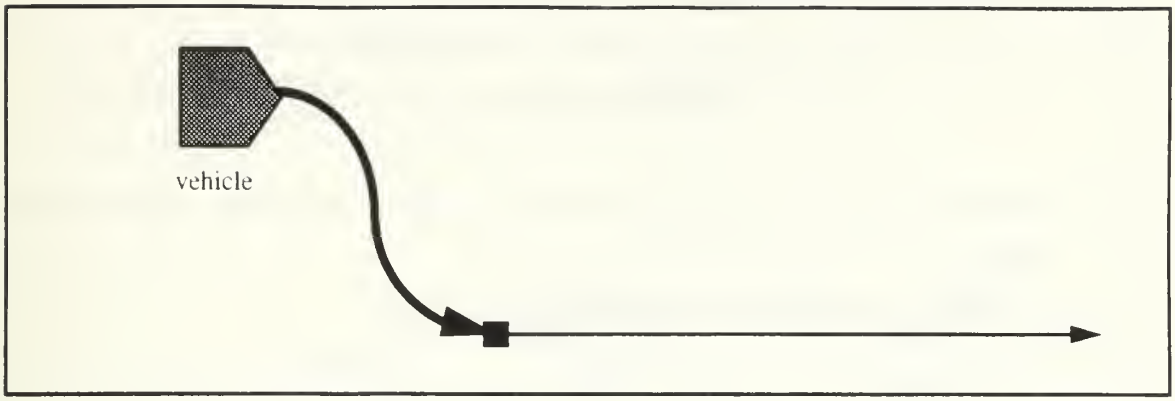

Figure 31: Example Of Forward Path Tracking

3. Move while Tracking a Backward Path (BPath)

Syntax: void BPath(double x, double y, double t)

void BPath(double x, double y, double t, double k)

void BPath(Config &q)

void BPath(Line &q)

void BPath(Circle &q)

Type: Sequential

Description: This function is utilized to accommodate the motion control and behavior of the path element defined as a BPath. See Figure 32. The robot as depicted in case one will follow the appropriate path form desired to the specified parameters. If the robot's position is beyond the specified parameters as depicted in case two. There are two possibilities. The first is the robot will come to a stop as soon as possible if there is no successor. The second is if there is a successor then the a path will be generated from that robot position.

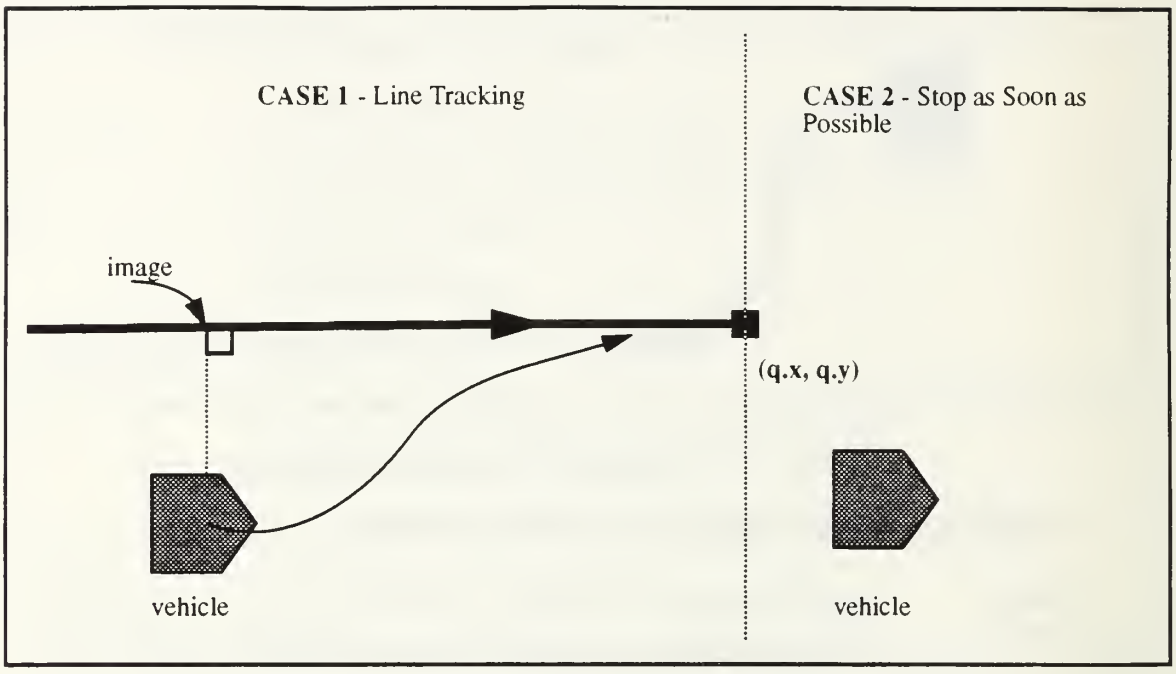

Figure 32: Examples Of Backward Path Tracking

4. Move While Tracking a Cubic Spiral (Posture)

Syntax: void Posture(double x, double y, double t)

void Posture(Cubic &q)

void Posture(Config &q)

Type: Sequential

Description: This function is utilized to accommodate the motion control and behavior of the path element defined as Posture. The robot as depicted in Figure 33, will generate and follow the cubic spiral path form from its current location to the specified parameters.

5. Leave Current Path Element (Skip)

Syntax: void Skip()

Type: Immediate

Description: This function is utilized to follow the next path element in the buffer. The current path element will be removed and the robot will project it's image to the next path element in the buffer.

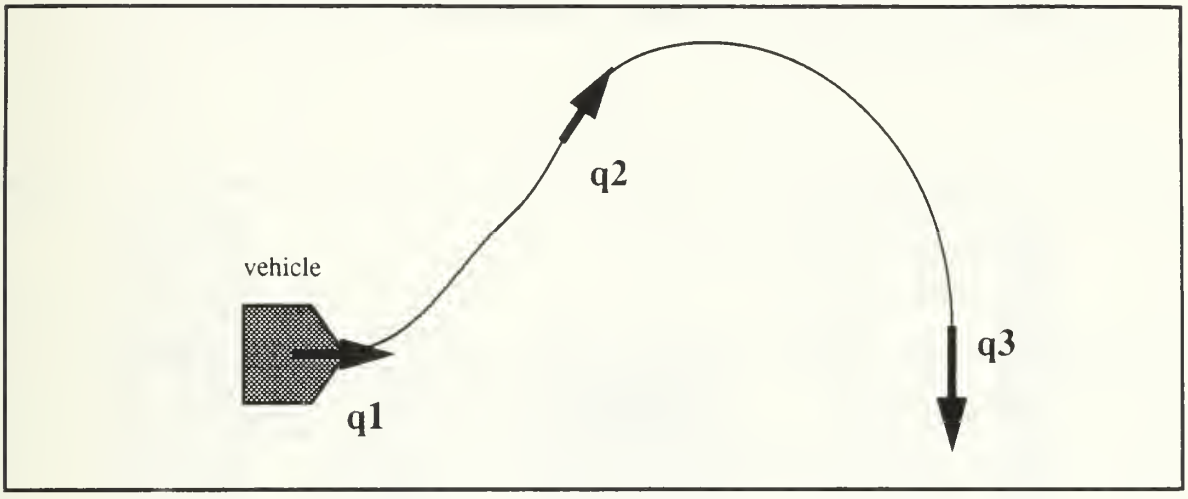

Figure 33: Example Of Posture To Posture Tracking

VII. SYSTEM ARCHITECTURE

The OOPS-MML system architecture schema is shown in Figure 34. The general concept is relatively still intact when compared to MML. Flow of control is from left to right and right to left movement is reserved for feedback. The remainder of this chapter is devoted to the idiosyncrasies of each module.

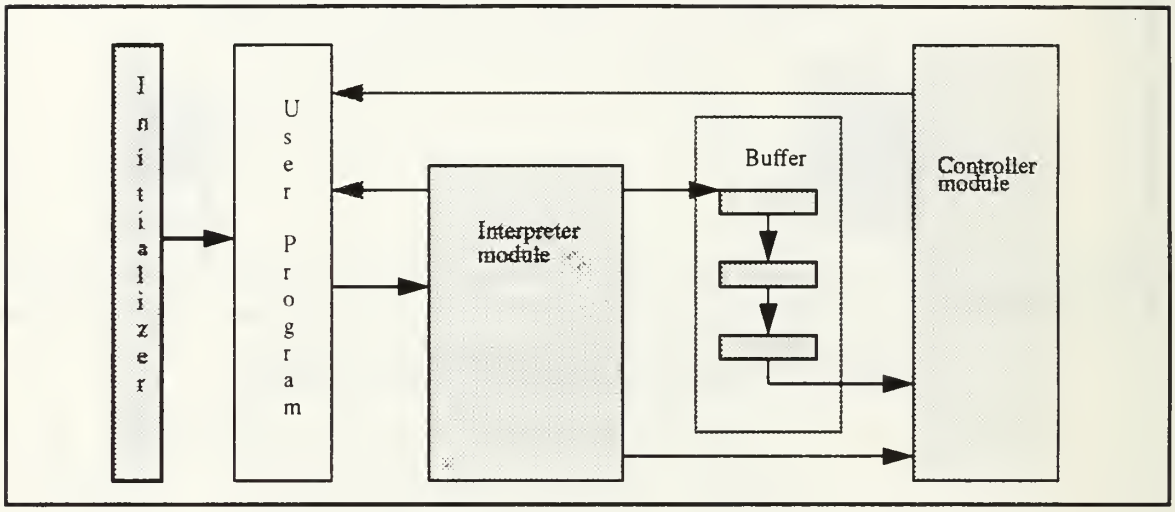

Figure 34: OOPS-MML System Architecture Schema

A. INITIALIZER MODULE

This module is the first stage in the flow of control. It is automatically started upon program execution. One of the functions of this module is to initialize global variables to default values. The other function, which has to written upon installation of the new board, will perform the initial setup and loading of registers.

B. USER PROGRAM MODULE

This module is the second stage in the flow of control. It is the detailed program written by the user to outline the intended behavior for the robot. The program is written utilizing the OOPS-MML function calls described in Chapter VI. To illustrate this, Figure 35 (a) shows an intended path and Figure 35 (b) shows the body of a user program. As you can see from the

program, path1 is defined in the path form of a *Line* and path2 in the path form of a *Circle*. The path elements chosen where *FPath* for the *Line* and *BPath* for the *Circle*. With these established, the only other functions required are *Set_Rob* and *End*, since default values could have been utilized instead of the additional commands. However, these commands are included to facilitate using this example throughout the rest of this chapter to gain insight into the process.

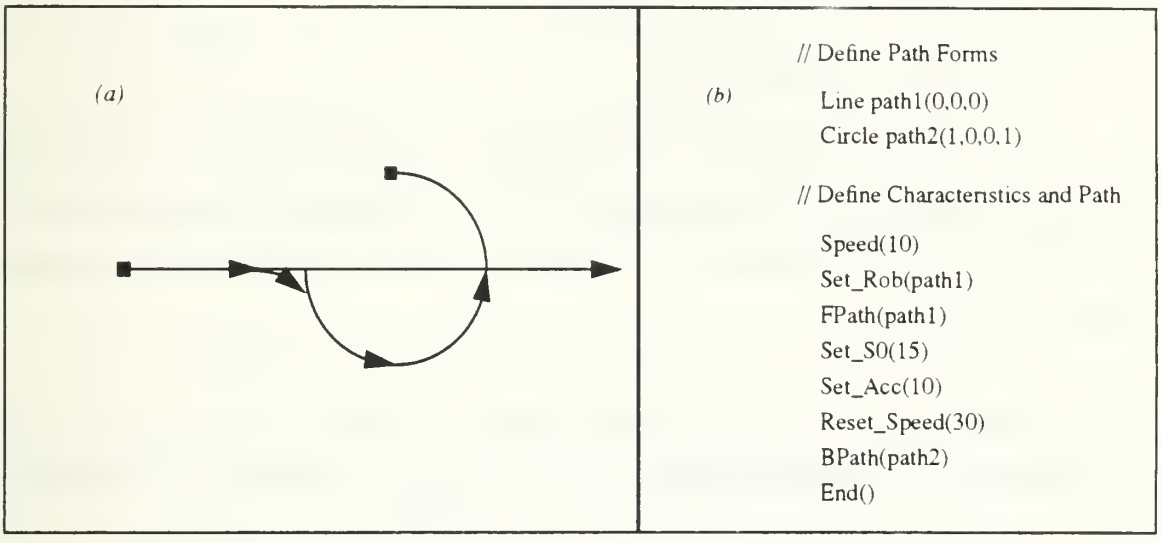

Figure 35: Example Of A Desired Behavior And The User Program

C. INTREPRETER MODULE

This module is the third stage in the flow of control and is the black box that incorporates all OOPS-MML function calls. It's primary function is to interpret the user program. The basis of interpretation is dependent on the temporal type of the specific function calls.

1. Immediate Functions

Functions of this temporal type are interpreted and then processed directly. All immediate functions, except for the *Skip()* function, are designed to instantaneously change one or more components in the *Vehicle* class object declared as *Robot*. For instance, the *Reset_Speed(sp)* function instantaneously sets the component for speed in *Robot*. The

controller module continually utilizes the current speed value in *Robot* to calculate motion. Thus, the robot accelerates or decelerates dependent upon the desired speed.

2. Sequential Functions

Functions of this temporal type are interpreted and then loaded into a command buffer. An example of the loaded command buffer for this user program is provided at the end of this subsection in Figure 37. For comprehension, an explanation of the background structure and construction process is necessary.

a. The Command Buffer

The command buffer is implemented as a linked list of buffer objects. Each buffer object is generic and is able to accommodate any command. This is accomplished by being able to cast a *Config* pointer to a *Config* object or any object below it in the inheritance schema, see Figure 36.

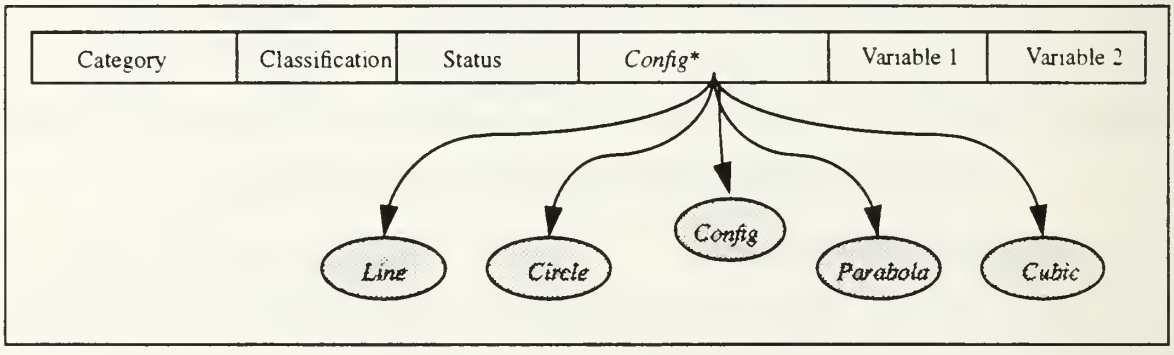

Figure 36: A Buffer Object

b. Interpretation of the Motion Category

Commands in this category define the four path elements. Once a path element command is received, it is loaded into the buffer with an incomplete status and the interpreter is placed in a pending status. Pending status freezes the buffer to any additional commands. In order for these statuses to be changed, a successor must be received and processed. When the successor is received, path elements are tested for compatibility. If they are found to be

compatible, appropriate actions are taken to process the requirements to obtain the specific transitory mode. Once this is accomplished, interpreter status is changed to non-pending, the transitory command is loaded onto the command buffer and then the previous command's status is changed to complete.

In the case of the user program in Figure 35, the successor to the FPath is a BPath. These are compatible and the transitory mode is TRM. Thus, they are tested for their transition behavior. These path forms have a common intersection point and will exhibit a bounded transition behavior by utilizing the path projection method. Once this is accomplished, the interpreter's status is changed to non-pending, the intersect command is loaded onto the buffer and then the *FPath's* command status is changed to complete. The intersect command houses the intersect of the path forms as a *Config* and the multiple for *s0* in the variable one slot.

c. Interpretation of the Parameter and Stationary Categories

The commands in the parameter category define characteristics of motion. The commands in the stationary category provide special motion capabilities. These commands can be interpreted relatively easily, since they are stand alone functions. Although, when they are loaded to the command buffer dependents solely on the status of the interpreter.

When the interpreter is in the non-pending status mode, commands are processed and loaded directly onto the command buffer. As can be surmised from the previous paragraphs, a status of non-pending is usually only realized at the start of the program before any motion commands.

When the interpreter is in the pending status mode. The commands are unable to be loaded into the command buffer, since it is frozen against additional commands. However, the requirement to process these commands still exists because they are in between path element commands. Although in order to process them, there is an additional requirement to maintain them in their proper sequential order. This is accomplished by utilization of a wait buffer. The wait buffer is identical to the command buffer but is inaccessible to the controller

module. This wait buffer houses the additional commands until they can be down loaded into the command buffer. This takes place after the interpreter's status has changed to non-pending, the transitory command has been placed on the buffer and the previous command's status has changed to complete. When the wait buffer is empty the interpreter's status is placed back in the pending mode.

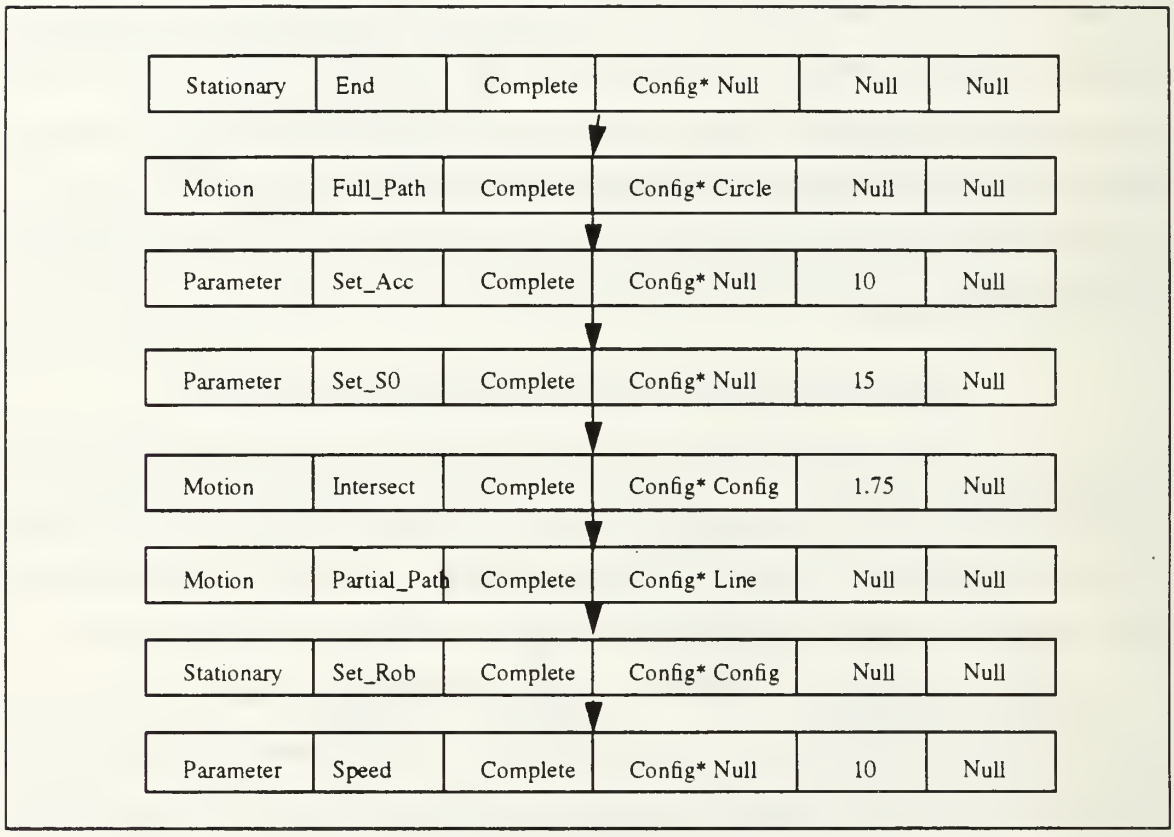

Figure 37: Example Of Loaded Buffer For User Program

D. CONTROLLER MODULE

This module is the fourth stage in the flow of control and is the black box that controls intended robotic behaviorisms. This is achieved by executing each command in the command buffer. Figure 38 *(a)* shows the logic behind the main controller module. This module continually reads the top of the command buffer and directs the flow of execution to the appropriate category module. Termination is realized in the event of an error or upon

execution of an *End* command. Figure 38 *(b)* shows the logic behind the main motion category module. This module directs the flow of execution to the specific motion classification module.

(a)

```
while ( ! TERMINATE )
    switch (Buffer.Top().Catagory() )
      case Parameter
        Execute_Parameter_Commands()
        break
      case Stationary
        Execute_Stationary_Commands()
        break
      case Motion
        Execute_Motion_Commands()
        break
      case Error
        TERMINATE = YES
        break
    end switch
```

(b)

```
switch (Buffer.Top().Class() )
  case INTERSECT
    Execute_Intersect_Command()
    break
  case FULL_PATH
    Execute_Full_Path_Command()
    break
  case PARTIAL_PATH
    Execute_Partial_Path_Command()
    break
  case CUBIC_SPIRAL
    Execute_Cubic_Spiral_Command()
    break
end switch
```

Figure 38: Example Of Logic For Execute Buffer And Execute Motion Commands

Each specific motion classification module basically provides two things. The first is the decision of whether or not to remove the command from the buffer. The second is the means to calculate a new Kappa to follow or converge onto the current path. An example for handling the intersect command is shown in Figure 39. If the transition point is reached the command is removed from the buffer. If not, the robot's image is projected onto the current path and a new kappa is calculated. Upon completion of this module the process will simply start over at the top. Actual code for the control module is in Appendix C.

```
Execute_Intersection_Command()

    // Check if at Transition Point

    if ( ! TRANSITION )
        // Not Ready: Utilize Value Only
        Current_Intersect = Buffer.Top().Command()

      // Get s0 Multiplier for Transition
        S0_Mult = Buffer.Top().Variable1

      // Calculate Image to Current Path
        Image = Current_Path->image()

      // Update Kappa to Converge to Path
        Update_Config(Current_Path, Image)

      // Check for Transition
        Transition = Image.Transition(Current_Intersect, S0_Mult,

    else
      // Remove Intersect Command from Buffer
        Current_Intersect = Buffer.Deque().Command()
```

Figure 39: Example Of Logic For Executing Intersect Commands

VIII. SUMMARY AND CONCLUSIONS

A. CONTRIBUTIONS OF WORK

The main thrust of this paper is to establish that an object-oriented approach for a mobile based robot control language is not only feasible but is highly desirable. This approach allowed us the following:

- The means to specify a complete abstract description of the path forms.
- The capabilities to build new path forms by composing or specializing existing ones.
- A way to describe path form interactions amongst themselves and or with the higher level abstractions such as path elements.
- The means to implement each of theses abstractions in a modular way.

Allowance of these supported the overall design goal of creating a language that would "stand the test of time" and directly supported the following design goals of:

- Easy implementation of refinements and or advancements.
- Maximum efficiency.
- Utilization of structures that provide maximum growth potential.

Implementation of OOPS-MML in the C++ language not only enhanced the realization of an object-oriented approach, it's idioms provided the means to directly support the remaining design goals. Specifically in:

- Creating a small set of clear, concise instructions.
- Allowing greater flexibility with parameters.
- Providing backward compatibility with existing code.

The thing lacking in this paper is the demonstration of the language with the real robot. This is because full implementation of OOPS-MML was predicated on the installation of a new central processing unit and mother board. Nevertheless there are two reasons for being confident that the methodology presented in this paper is absolutely sound. The first is given the fact that currently a control system utilizing path control by curvature has been realized by patch working the current MML. This has proven quite successful in tracking path elements and is the same system employed in OOPS-MML. Although with the exception of the

algorithms being written in C. The second reason is due to the intense testing on the concepts [5] and simulation performance of OOPS-MML.

B. FUTURE RESEARCH

The power and beauty of OOPS-MML is that it was conceived and designed to be a "foundation for growth". Although in order to realize this, the first step is to fully implement it upon arrival of the new components. This can be done preferably by converting the existing low level modules to C++ or if need be, by interfacing with the existing C code. Once fully implemented though, growth potential and future research are limitless due to the abstract nature of the foundation. The remaining of the chapter elaborates a few.

1. Robot Agility

Path planning is relatively simple for a human being by virtue of their vast agility. Although Yamabico may never be able to vertical climb, jump or instantaneously move sideways, it is capable of following an endless supply of path elements and combinations thereof. Future research for greater agility could be realized by increasing the existing permissible combinations such as *Parabola* to *Circle* or vice versa. Creation of different path forms is also another avenue, such as an *Ellipse* of a *Hyperbola*.

2. Navigator

The navigator could provide a dead reckoning position from known land marks. This information could either be used by other peripherals or provide for automatic *Vehicle* object component correction.

3. Mission Planner

The mission planner could be able to generate a optimum paths based upon specific missions such as mail delivery. It could also be dynamic and provide for obstacle avoidance or possible revisions to the optimum path based on current availability and information.

APPENDIX A: C++ CODE OF CLASSES

```
#ifndef Coord_H
#define Coord_H

#include <iostream.h>

class Coordinate
  {
    public:

    // Structure
      double _X;
      double _Y;

    // Constructors
      Coordinate();
      Coordinate(double, double);
      Coordinate(const Coordinate&);

    // Destructor
      virtual ~Coordinate() { }

    // Operators
      Coordinate& operator=(const Coordinate&);

    // Friend Functions
      friend ostream &operator<<(ostream&, Coordinate&);
      friend istream &operator>>(istream&, Coordinate&);

    // Inline Mutators
      void set_x(double x) { _X = x; }
      void set_y(double y) { _Y = y; }
      void set(double x, double y) { _X = x; _Y = y; }

    // Inline Accessors
      double X() const { return _X; }
      double Y() const { return _Y; }

    // Utilities
      virtual double distance(Coordinate&, Coordinate&);
};
#endif
```

```
#include "coord.h"
#include <math.h>
// Constructors
Coordinate :: Coordinate()
  {
    _X = 0;
    _Y = 0;
  }

Coordinate :: Coordinate(double x, double y)
  {
    _X = x;
    _Y = y;
  }

Coordinate :: Coordinate(const Coordinate &p)
  {
    _X = p._X;
    _Y = p._Y;
  }

// Operators
Coordinate&
Coordinate :: operator=(const Coordinate &p)
  {
    _X = p._X;
    _Y = p._Y;
    return *this;
  }

// Friend Functions
ostream &operator<<(ostream &strm, Coordinate &p)
  {
    return strm << "(" << p._X << "," << p._Y << ")";
  }

istream &operator>>(istream &strm, Coordinate &p)
  {
    double x,y;
    char   c;

    if ( strm >> x )
      {
                if ( strm >> c )
                  {
                    if ( c == ',' )
                            {
                              if ( strm >> y )
                                {
                                        p.set(x, y);
                                        return strm;
                                }
                            }
                  }
      }
    strm.clear( ios::badbit | strm.rdstate() );
    return strm;
  }

// Utilities
double
Coordinate :: distance(Coordinate &p1, Coordinate &p2)
  {
    float dx = p1._X - p2._X;
    float dy = p1._Y - p2._Y;
    return sqrt( dx*dx + dy*dy );
  }
```

```
ifndef Config_H
#define Config_H

#include "coord.h"
#include <iostream.h>

class Line;
class Circle;
class Parabola;

class Config : public Coordinate
  {
    public :

    // Structure
      double _Theta;
      double _Kappa;

    // Constructors
      Config();
      Config(double, double, double);
      Config(double, double, double, double);
      Config(double, double, double, double, double);
      Config(const Config&);
      Config(const Config*);

    // Destructor
      virtual ~Config() { }

    // Operators
      Config& operator=(const Config&);
      friend ostream &operator<<(ostream&, Config&);
      friend ostream &operator<<(ostream&, Config*);

    // Inline Mutator
      void set_theta(double t) { _Theta = t; }
      void set_kappa(double k) { _Kappa = k; }

    // Inline Accessor
      double Theta() { return _Theta; }
      double Kappa() { return _Kappa; }

    // Utilities
      double distance(Config&, Config&);

    // Virual Fundtions
      virtual void   set(Config*);
      virtual Config intersects(Line&);
      virtual Config intersects(Circle&);
      virtual Config intersects(Parabola&);
      virtual Config image();
      virtual int Transition(Config&, double);
      virtual int Complete(Config&);
      virtual Config Project_image();
      virtual void   Project_Start_image(Config&, double);

      double Project_Delta_dist(Config&);
      double Project_Kappa(Config&, double);
      double Project_Delta_theta(double, double);
      friend double Project_Update_config(Config&, Config&);
      int Project_Transition(Config&);
 };

#endif
```

```
include "config.h"
#include "line.h"
#include "vehicle.h"
#include "util.h"
#include <math.h>
#include <iomanip.h>

// External Reference for Image Distance from Next Intersection
  extern double PDist;

// external Reference for Projection
  extern Vehicle Projection;
  extern Vehicle Robot;

// Constructors
Config:: Config()
 : Coordinate()
 {
   _Theta = 0;
   _Kappa = 0;
 }

Config :: Config(double x, double y, double t)
 : Coordinate(x,y)
 {
   _Theta = t;
   _Kappa = 0;
 }

Config :: Config(double x, double y, double t, double k)
 : Coordinate(x, y)
 {
   _Theta = t;
   _Kappa = k;
 }

Config :: Config(const Config &temp)
 : Coordinate(temp._X, temp._Y)
 {
   _Theta = temp._Theta;
   _Kappa = temp._Kappa;

 }

Config :: Config(const Config *temp)
 : Coordinate(temp->_X, temp->_Y)
 {
   _Theta = temp->_Theta;
   _Kappa = temp->_Kappa;
 }


// Operators
Config&
Config :: operator=(const Config &c)
 {
   _X = c._X;
   _Y = c._Y;
   _Theta = c._Theta;
   _Kappa = c._Kappa;
   return *this;
 }


// Friend Functions
ostream &operator<<(ostream &strm, Config &p)
 {
   return strm << "( " << setprecision(2) << p._X
```

```
                        << ", " << setprecision(2) << p._Y
                        << ", " << setprecision(1) << p._Theta * RAD_DEG
                        << ", " << setprecision(2) << p._Kappa
                        << " )";
  }

ostream &operator<<(ostream &strm, Config *p)
  {
    return strm << "( " << setprecision(2) << p->_X
                        << ", " << setprecision(2) << p->_Y
                        << ", " << setprecision(1) << p->_Theta * RAD_DEG
                        << ", " << setprecision(2) << p->_Kappa
                        << " )";
  }


// Utilities
void
Config :: set(Config *v)
  {
    _X     = v->_X;
    _Y     = v->_Y;
    _Theta = v->_Theta;
    _Kappa = v->_Kappa;
  }

double
Config :: distance(Config &c1, Config &c2)
  {
    return ((c1._Y - c2._Y) * cos(c2._Theta) -
                (c1._X - c2._X) * sin(c2._Theta));
  }

int
Config :: Transition(Config &Intersect, double Mult)
  {
    double Image_dist = Coordinate::distance( *this, Intersect);
    double Trans_dist = Mult * Robot.s0();
    if ( fabs(Image_dist) <= Trans_dist)
      {
                return 1;
      }
    else
      {
                return 0;
      }
  }

int
Config :: Complete(Config &Stop_Config)
  {
    double Epsilon    = 0.5;
    double Image_dist = Coordinate::distance( *this, Stop_Config);

    if ( fabs(Image_dist) <= Epsilon)
                return 1;
    else
                return 0;
  }


Config
Config :: intersects(Line &l)
  {
    cout << "Entered config::intersects line \n" << flush;
    return *this;
  }
```

```
Config
Config :: intersects(Circle &c)
  {
    cout << "Entered config::intersects circle \n" << flush;
    return *this;
  }

Config
Config :: intersects(Parabola &p)
  {
    cout << "Entered config::intersects Parabola \n" << flush;
    return *this;
  }

Config
Config :: image()
  {
    cout << "Entered config::image \n" << flush;
    return *this;
  }

Config
Config :: Project_image()
  {
    cout << "Entered Config::Project_image \n" << flush;
    return *this;
  }

void
Config :: Project_Start_image(Config &I, double M)
  {
    cout << "Entered Config::Project_Start_image \n" << flush;
  }


int
Config :: Project_Transition(Config &Current_Path)
  {
    double image_dist = Coordinate::distance(*this, Current_Path);

    if ( ( fabs(PDist) < 0.01) && (fabs(Projection._Theta - _Theta) < 0.0175) )
      {
              return 1;
      }
    else
      {
              return 0;
      }
  }
```

```
#ifndef Vehicle_H
#define Vehicle_H

#include "config.h"
#include "util.h"
#include <math.h>
#include <iostream.h>
#include <iomanip.h>

class Vehicle : public Config
  {
    public :

    // Structure
      double _Speed;
      double _Omega;
      double _S0;
      double _L_Accel;
      double _R_Accel;

    // Constructors
      Vehicle();
      Vehicle(double,double,double,double);
      Vehicle(double,double,double,double,double);
      Vehicle(Vehicle&);
      Vehicle(Vehicle*);

    // Destructor
      virtual ~Vehicle() { }

    // Operators
      Vehicle &operator=(Vehicle&);
      Vehicle &operator=(Config*);

    // Inline Mutators
      void set_Speed(double s)   { _Speed = s;   }
      void set_Omega(double o)   { _Omega = o;   }
      void set_S0(double s)      { _S0 = s;      }
      void set_L_Accel(double a) { _L_Accel = a; }
      void set_R_Accel(double r) { _R_Accel = r; }

    // Inline Accessors
      double Speed()   { return _Speed; }
      double Omega()   { return _Omega; }
      double s0()      { return _S0;    }
      double L_Accel() { return _L_Accel; }
      double R_Accel() { return _R_Accel; }

    // Friend Functions
      friend ostream &operator<<(ostream&, Vehicle&);

    // Utilities
      double Delta_dist(Config&);
      double Kappa_Value(Config&, double);
      double Delta_theta(double, double);
      friend void Update_config(Config&, Config&);

  };
#endif
```

```
#include "vehicle.h"

// Global Variables used for Control Phase
  extern Vehicle Robot;
  double delta_time = .01;

// Global Variables used for Interpet Phase
  extern Vehicle Projection;
  double PDist;


// Constructors
Vehicle :: Vehicle()
  : Config()
  {
    _Speed   = 0;
    _Omega   = 0;
    _S0      = 15;
    _L_Accel = 20;
    _R_Accel = 10;
  }

Vehicle :: Vehicle(double x, double y, double t, double k)
  : Config(x,y,t,k)
  {
    _Speed   = 30;
    _Omega   = 30 * k;
    _S0      = 15;
    _L_Accel = 20;
    _R_Accel = 10;
  }

Vehicle :: Vehicle(double x, double y, double t, double k, double s)
  : Config(x,y,t,k)
  {
    _Speed   = s;
    _Omega   = s * k;
    _S0      = 15;
    _L_Accel = 20;
    _R_Accel = 10;
  }

Vehicle :: Vehicle(Vehicle &t)
  : Config(t._X, t._Y, t._Theta, t._Kappa)
  {
    _Speed   = t._Speed;
    _Omega   = t._Omega;
    _S0      = t._S0;
    _L_Accel = t._L_Accel;
    _R_Accel = t._R_Accel;
  }

Vehicle :: Vehicle(Vehicle *t)
  : Config(t->_X, t->_Y, t->_Theta, t->_Kappa)
  {
    _Speed   = t->_Speed;
    _Omega   = t->_Omega;
    _S0      = t->_S0;
    _L_Accel = t->_L_Accel;
    _R_Accel = t->_R_Accel;
  }

Vehicle&
Vehicle :: operator=(Vehicle &v)
  {
    _X       = v._X;
    _Y       = v._Y;
    _Theta   = v._Theta;
```

```
    _Kappa   = v._Kappa;
    _Speed   = v._Speed;
    _Omega   = v._Omega;
    _S0      = v._S0;
    _L_Accel = v._L_Accel;
    _R_Accel = v._R_Accel;
    return *this;
  }


Vehicle&
Vehicle :: operator=(Config *v)
  {
    _X       = v->_X;
    _Y       = v->_Y;
    _Theta   = v->_Theta;
    _Kappa   = v->_Kappa;
    return *this;
  }


// Friend Functions
ostream &operator<<(ostream &strm, Vehicle &v)
  {
    return strm << "( " << setprecision(3) << v._X
                        << ", " << setprecision(3) << v._Y
                        << ", " << setprecision(3) << v._Theta * RAD_DEG
                        << ", " << setprecision(2) << v._Kappa
                        << ", " << setprecision(1) << v._Speed
                        << ", " << setprecision(2) << v._Omega
                        << ", " << setprecision(2) << v._S0
                        << ", " << setprecision(2) << v._L_Accel
                        << ", " << setprecision(2) << v._R_Accel
                        << " )";
  }


// Ultilities
double Delta_dist(Config &path)
  {
    double dist;

  // Calculate Distance to Path
    dist = ( -(Robot._X - path._X) * (path._Kappa
                * (Robot._X - path._X) + 2 * sin(path._Theta))
                - (Robot._Y - path._Y) * (path._Kappa
                * (Robot._Y - path._Y) - 2 * cos(path._Theta)))
                / (1 + sqrt(square(path._Kappa * (Robot._X - path._X)
                + sin(path._Theta)) + square(path._Kappa * (Robot._Y - path._Y)
                - cos(path._Theta))));
  // Return Distance to Path
    return dist;

  }

double Kappa_Value(Config &image, double dist)
  {
    double k = 1.0 / Robot._S0;
    double A = 3.0 * k;
    double B = A * k;
    double C = B * k / 3.0;
    double K = -((A * (Robot._Kappa - image._Kappa))
                        + (B * norm(Robot._Theta - image._Theta))
                        + (C * min_range(dist, Robot._S0)));
  // Return Kappa required to get on Image
    return K;
  }
```

```
void Update_config(Config &path, Config &image)
 {
   double Delta_Theta;
   double Delta_Dist, Delta_Dist1;
   double Dist;
   double New_Kappa;
   double epsilon = 0.00001;

 // Calculate Change of Distance the Robot Traveled
   Delta_Dist = Delta_Dist1 = delta_time * Robot.Speed();

 // Calculate Distance to path
   Dist = Delta_dist(image);

 // Calculate Kappa to Path
   New_Kappa = Robot._Kappa + Kappa_Value(image, Dist) * Delta_Dist;

 // Calculate Change in Theta Required to Move onto Path
   Delta_Theta = Delta_Dist * New_Kappa;

 // Check if Delta Theta is not Zero
   if (!(Delta_Theta == 0))
     Delta_Dist1 = Delta_Dist * (sin(Delta_Theta/2) / (Delta_Theta/2));

 // Calculate Vehicle X Position
   Robot._X += (Delta_Dist1 * cos(Robot._Theta + (Delta_Theta/2.0)));

 // Calculate Vehicle Y Position
   Robot._Y += (Delta_Dist1 * sin(Robot._Theta + (Delta_Theta/2.0)));

 // Calculate Vehicle Theta Value
   Robot._Theta  = norm(Robot._Theta + Delta_Theta);

 // Calculate Vehicle Kappa Value
   Robot._Kappa  = New_Kappa;

 // Calculate Vehicle Omega Value
   Robot._Omega  = Robot.Speed() * New_Kappa;

 }

void Project_Delta_dist(Config &path)
 {
 // Calculate Distance to Path
   PDist =( -(Projection._X - path._X) * (path._Kappa
               * (Projection._X - path._X) + 2 * sin(path._Theta))
               - (Projection._Y - path._Y) * (path._Kappa
               * (Projection._Y - path._Y) - 2 * cos(path._Theta)))
               / (1 + sqrt(square(path._Kappa * (Projection._X - path._X)
               + sin(path._Theta)) + square(path._Kappa * (Projection._Y - path._Y)
               - cos(path._Theta))));

 }


double Project_Kappa(Config &image, double dist)
 {
   double k = 1.0 / Projection._S0;
   double A = 3.0 * k;
   double B = A * k;
   double C = B * k / 3.0;
   double K = -((A * (Projection._Kappa - image._Kappa))
                       + (B * norm(Projection._Theta - image._Theta))
                       + (C * min_range(dist, Projection._S0)));
   return K;
 }
```

```
double Project_Update_config(Config &path, Config &image)
 {
   double Delta_Theta;
   double Delta_Dist, Delta_Dist1;
   double Dist;
   double New_Kappa;
   double epsilon = 0.00001;

 // Calculate Change of Distance the Robot Traveled
//    Delta_Dist = Delta_Dist1 = delta_time * Projection.Speed();
   Delta_Dist = 1;
 // Calculate Distance to path
   Project_Delta_dist(image);

 // Calculate Kappa to Path
   New_Kappa = Projection._Kappa + Project_Kappa(image, PDist) * Delta_Dist;

 // Calculate Change in Theta Required to Move onto Path
   Delta_Theta = Delta_Dist * New_Kappa;

 // Check if Delta Theta is not Zero
   if (!(Delta_Theta == 0))
     Delta_Dist1 = Delta_Dist * (sin(Delta_Theta/2) / (Delta_Theta/2));

 // Calculate Vehicle X Position
   Projection._X += (Delta_Dist1 * cos(Projection._Theta + (Delta_Theta/2.0)));

 // Calculate Vehicle Y Position
   Projection._Y += (Delta_Dist1 * sin(Projection._Theta + (Delta_Theta/2.0)));

 // Calculate Vehicle Theta Value
   Projection._Theta  = norm(Projection._Theta + Delta_Theta);

 // Calculate Vehicle Kappa Value
   Projection._Kappa  = New_Kappa;

 // Calculate Vehicle Omega Value
   Projection._Omega  = Projection.Speed() * New_Kappa;

   return Delta_Dist;
 }
```

```
#ifndef Line_H
#define Line_H

#include "config.h"

class Circle;
class Parabola;


class Line : public Config
 {
  public:

  // Structure
    Coordinate _P2;
    double _A;
    double _B;
    double _C;


  // Constructors
    Line();
    Line(double, double, double);
    Line(double, double, double, double);
    Line(Coordinate&, Coordinate&);
    Line(const Line&);


  // Destructor
    virtual ~Line() { }


  // Operators
    Line&    operator=(const Line&);
    friend ostream& operator<<(ostream&, Line&);

  // Inline Accessors
    double A() const { return _A; }
    double B() const { return _B; }
    double C() const { return _C; }
    Coordinate P2()  { return _P2;}

  // Utilities
    Config intersects(Line&);
    Config intersects(Circle&);
    Config intersects(Parabola&);
    Config image();
    Config Project_image();
    void   Project_Start_image(Config&, double);

 };

#endif
```

```
#include "line.h"
#include "circle.h"
#include "parabola.h"
#include "vehicle.h"
#include "util.h"
#include <math.h>

extern Vehicle Robot;
extern Vehicle Projection;

// Constructors
Line :: Line()
 : Config()
 {
   _P2 = Coordinate();
   _A  = 0;
   _B  = 0;
   _C  = 0;
 }

Line :: Line(double x, double y, double t)
 : Config(x, y, t, 0)
 {
   double t1 = DEG_RAD * t;
   _P2 = Coordinate( (x + 10 * cos(t1)), (y - 10 * sin(t1)) );
   _A  = _P2._Y - _Y;
   _B  = x - _P2._X;
   _C  = (_P2._X * y) - (x * _P2._Y);
 }

Line :: Line(double x, double y, double t, double k)
 : Config(x, y, t * DEG_RAD, k)
 {
   double t1 = DEG_RAD * t;
   _P2 = Coordinate( (x + 10 * cos(t1)), (y - 10 * sin(t1)) );
   _A  = _P2._Y - y;
   _B  = x - _P2._X;
   _C  = (_P2._X * y) - (x * _P2._Y);
 }

Line :: Line(Coordinate &p1, Coordinate &p2)
 : Config(p1._X, p1._Y, 0, 0)
 {
   _P2 = p2;
   _A  = p2._Y - p1._Y;
   _B  = p1._X - p2._X;
   _C  = (p2._X * p1._Y) - (p1._X * p2._Y);
   _Theta = atan2(p2._Y - p1._Y, p2._X - p1._X);
 }

Line :: Line(const Line &l)
 : Config(l._X, l._Y, l._Theta, l._Kappa)
 {
   _P2 = l._P2;
   _A  = l._A;
   _B  = l._B;
   _C  = l._C;
 }

// Operators
Line&
Line :: operator=(const Line &L)
 {
   _X = L._X;
   _Y = L._Y;
   _Theta = L._Theta;
   _Kappa = L._Kappa;
   _P2 = L._P2;
```

```
    _A  = L._A;
    _B  = L._B;
    _C  = L._C;
    return *this;
  }


// Friend Functions
ostream &operator<<(ostream &strm, Line &p)
  {
    return strm << "( "      << p._X
                            << ", "     << p._Y
                            << " )( "   << p.P2()._X
                            << ", "     << p.P2()._Y
                            << " ) "    << p._Theta * RAD_DEG;
                            << ",  0 = " << p.A()
                            << "x + "   << p.B()
                            << "y +"    << p.C();
  }


// Utilities
Config
Line :: intersects(Line &L)
  {
    double x;
    double y;
  // Calculate X Intersect
    x = (-cos(L._Theta)*((_X*sin(_Theta))-(_Y*cos(_Theta)))
                +cos(_Theta)*((L._X*sin(L._Theta))-(L._Y * cos(L._Theta))))
                /sin(L._Theta - _Theta);

  // Calculate Y Intersect
    y = ( sin(_Theta)*((L._X*sin(L._Theta))-(L._Y*cos(L._Theta)))
                -sin(L._Theta)*((_X*sin(_Theta))-(_Y*cos(_Theta))))
                /sin(L._Theta - _Theta);

  // Return Intersect Posture
    return Config(x, y, L._Theta);
  }


Config
Line :: intersects(Circle &C)
  {
  // Local Variables
    double    A_dist;
    double    B_dist;
    double    Phi;
    Coordinate Image;
    Coordinate Intersect1;
    Coordinate Intersect2;
    Config    Intersect;

  // Calculate distance from center to a perpendicular point on line
    A_dist = (C._Center._Y - _Y) * cos(_Theta) - (C._Center._X - _X) * sin(_Theta);

  // Calculate
    Image._X = C.Center()._X + A_dist * sin(_Theta);
    Image._Y = C.Center()._Y - A_dist * cos(_Theta);

  // Calculate distance from Perpendicular Point to Intersect Points
    B_dist = sqrt(fabs(square(C.Radius()) - square(A_dist)));

  // Calculate Upwind and Downwind Intersect Points
    Intersect1._X = Image._X + B_dist * cos(_Theta);
    Intersect1._Y = Image._Y + B_dist * sin(_Theta);
```

```
    Intersect2._X = Image._X - B_dist * cos(_Theta);
    Intersect2._Y = Image._Y - B_dist * sin(_Theta);

  // Determine Correct Intersect Point between Upwind and Downwind
    if (((Intersect1._X - _X) * cos(_Theta)
              +(Intersect1._Y - _Y) * sin(_Theta))
             <((Intersect2._X - _X) * cos(_Theta)
              +(Intersect2._Y - _Y) * sin(_Theta)))
      {
              Intersect._X = Intersect1._X;
              Intersect._Y = Intersect1._Y;
             }
    else
             {
              Intersect._X = Intersect2._X;
              Intersect._Y = Intersect2._Y;
             }

  // Calculate Circle Angle of Intersect Point
    Phi = atan2(Intersect._Y - C._Center._Y, Intersect._X - C._Center._X);

  // Calculate Angle on Circle
    Intersect._Theta = norm( Phi + HPI * (C._Kappa / fabs(C._Kappa)));

  // Return Intersect of Line to Circle
    return Intersect;

  }

Config
Line :: intersects(Parabola &P)
  {
    double A, B, C;
    double Dist1, Dist2, I1_Dist, I2_Dist;
    Config Intersect;
    Coordinate I1, I2;

    A = (square(sin(_Theta)) * square(cos(P._Theta)))
     + (square(sin(P._Theta)) * square(cos(_Theta)))
     - (2.0 * sin(_Theta) * cos(_Theta) * sin(P._Theta) * cos(P._Theta))
     - 1.0;

    B = (2.0
     * (((P._X - _X) * sin(P._Theta)
     * ((sin(P._Theta) * cos(_Theta)) - (cos(P._Theta) * sin(_Theta))) )
     + ( (P._Y - _Y) * cos(P._Theta)
     * ((cos(P._Theta) * sin(_Theta)) - (sin(P._Theta) * cos(_Theta))) )
     + (P.Focus()._X - _X) * cos(_Theta)
     + (P.Focus()._Y - _Y) * sin(_Theta)));


    C = ( square(_Y - P._Y) * square(cos(P._Theta)))
     + ( square(_X - P._X) * square(sin(P._Theta)))
     + ( 2.0 * cos(P._Theta) * sin(P._Theta)
     * ( _X * P._Y + P._Y * _X - _X * _Y - P._X * P._Y))
     - square(_X - P.Focus()._X)
     - square(_Y - P.Focus()._Y);


    if ( A == 0 )
      {
              if ( B == 0)
               {
                Intersect._X = _X + C * cos(_Theta);
                Intersect._Y = _Y + C * sin(_Theta);
               }
              else
               {
```

```
            Dist1 =  -C / B;
                  Intersect._X = _X + Dist1 * cos(_Theta);
                  Intersect._Y = _Y + Dist1 * sin(_Theta);
                }
      }
    else
      {
            Dist1 = root_pos(A,B,C);
            Dist2 = root_neg(A,B,C);
            I1._X = _X + Dist1 * cos(_Theta);
            I1._Y = _Y + Dist1 * sin(_Theta);
            I2._X = _X + Dist2 * cos(_Theta);
            I2._Y = _Y + Dist2 * sin(_Theta);

            I1_Dist = ( I1._X - _X ) * cos(_Theta)
                      + ( I1._Y - _Y ) * sin(_Theta);

            I2_Dist = ( I2._X - _X ) * cos(_Theta)
                      + ( I2._Y - _Y ) * sin(_Theta);

            if ( Dist1 < Dist2 )
              {
                Intersect._X = I1._X;
                Intersect._Y = I1._Y;
              }
            else
              {
                Intersect._X = I2._X;
                Intersect._Y = I2._Y;
              }
      }
    return Intersect;
  }


Config
Line :: image()
  {
    float  Dist;
    double Image_x;
    double Image_y;

  // Calculate Perpendicular Distance on Path to Vehicle
    Dist = Config::distance(Robot, *this);

  // Calculate Perpendicular X Position on Line
    Image_x = Robot._X + Dist * sin(_Theta);

  // Calculate Perpendicular Y Position on Line
    Image_y = Robot._Y - Dist * cos(_Theta);

  // Return Image
    return Config(Image_x, Image_y, _Theta);
  }
```

```
Config
Line :: Project_image()
 {
   float  Dist;
   double Image_x;
   double Image_y;

 // Calculate Perpendicular Distance on Path to Vehicle
   Dist = Config::distance(Projection, *this);

 // Calculate Perpendicular X Position on Line
   Image_x = Projection._X + Dist * sin(_Theta);

 // Calculate Perpendicular Y Position on Line
   Image_y = Projection._Y - Dist * cos(_Theta);

 // Return Image
   return Config(Image_x, Image_y, _Theta);
 }

void
Line :: Project_Start_image(Config &Intersect, double Mult)
 {

 // Calculate Perpendicular X Position on Line
   Projection._X = Intersect._X - Mult * Projection.s0() * cos(_Theta);

 // Calculate Perpendicular Y Position on Line
   Projection._Y = Intersect._Y - Mult * Projection.s0() * sin(_Theta);

 // Set Projection Theta to Line Theta
   Projection._Theta = _Theta;
 }
```

```
#ifndef Circle_H
#define Circle_H
```

```
#include "config.h"

class Line;

class Circle : public Config
  {
   public:

    // Structure
      Coordinate  _Center;
      double _Radius;

    // Constructors
      Circle();
      Circle(double, double, double);
      Circle(double, double, double, double);
      Circle(const Circle&);

    // Destructors
      ~Circle(void) { }

    // Operators
      Circle& operator=(const Circle&);
      friend ostream& operator<<(ostream&, Circle&);


    // Inline Accessors
      Coordinate Center() { return _Center; }
      double Radius() { return _Radius; }

    // Utilities
      Config intersects(Line&);
      Config intersects(Circle&);
      Config image();
      Config Project_image();
      void   Project_Start_image(Config&, double);
  };

#endif
```

```
include "circle.h"
#include "line.h"
```

```
#include "vehicle.h"
#include "util.h"
#include <math.h>

extern Vehicle Robot;
extern Vehicle Projection;

// Constructors
Circle :: Circle()
  :  Config()
  {
    _Center = Coordinate(0,0);
    _Radius = 0;
  }

Circle :: Circle(double x, double y, double r)
  :  Config(0,0,0,1/r)
  {
    _Center = Coordinate(x, y);
    _Radius = r;
    _X = x + r * cos(HPI);
    _Y = y - r * sin(HPI);
  }

Circle :: Circle(double x, double y, double t, double k)
  :  Config(x, y, t * DEG_RAD, k)
  {
    _Radius = 1.0/k;
    _Center._X = x - _Radius * sin(DEG_RAD * t);
    _Center._Y = y + _Radius * cos(DEG_RAD * t);
  }

Circle :: Circle(const Circle &c)
  :  Config(c._X, c._Y, c._Theta, c._Kappa)
  {
    _Center = c._Center;
    _Radius = c._Radius;
  }

// Operators
Circle&
Circle :: operator=(const Circle &c)
  {
    _X = c._X;
    _Y = c._Y;
    _Theta  = c._Theta;
    _Kappa  = c._Kappa;
    _Center = c._Center;
    _Radius = c._Radius;
    return *this;
  }


// Friend Functions
ostream &operator<<(ostream &strm, Circle &c)
  {
    return strm << "( " << c._Center._X
                        << ", " << c._Center._Y
                        << " ) "<< c._Radius;
  }




// Utilities
Config
Circle :: intersects(Line &L)
```

```
{
// Local Variables
  double    A_dist;
  double    B_dist;
  Coordinate Image;
  Coordinate Intersect1;
  Coordinate Intersect2;
  Config    Intersect;

// Calculate distance from center to a perpendicular point on line
  A_dist = (_Center._Y - L._Y) * cos(L._Theta) - (_Center._X - L._X) * sin(L._Theta);

// Calculate
  Image._X = _Center._X + A_dist * sin(L._Theta);
  Image._Y = _Center._Y - A_dist * cos(L._Theta);

// Calculate distance from Perpendicular Point to Intersect Points
  B_dist = sqrt(fabs(square(_Radius) - square(A_dist)));

// Calculate Upwind and Downwind Intersect Points
  Intersect1._X = Image._X + B_dist * cos(L._Theta);
  Intersect1._Y = Image._Y + B_dist * sin(L._Theta);

  Intersect2._X = Image._X - B_dist * cos(L._Theta);
  Intersect2._Y = Image._Y - B_dist * sin(L._Theta);

// Determine Correct Intersect Point between Upwind and Downwind
  if (((Intersect1._X - _X) * cos(L._Theta)
          +(Intersect1._Y - _Y) * sin(L._Theta))
         <((Intersect2._X - _X) * cos(L._Theta)
         +(Intersect2._Y - _Y) * sin(L._Theta)))
    {
         Intersect._X = Intersect2._X;
         Intersect._Y = Intersect2._Y;
        }
  else
        {
         Intersect._X = Intersect1._X;
         Intersect._Y = Intersect1._Y;
        }

// Set Intersect Theta to Line theta
  Intersect._Theta = L._Theta;

// Return Intersect Config
  return Intersect;

 }

Config
Circle :: intersects(Circle &C2)
 {
  double K, A, B, C, L, A1, A2, A_ref;
  Config intersect_neg;
  Config intersect_pos;
  Config Intersect;

// Check for Circles in same X Coordinates
  if (_Center._X == C2._Center._X)
   {

        K = ( square(_Radius) - square(C2._Radius)
          + ( square(C2._Center._Y) - square(_Center._Y)))
          / ( 2 * ( C2._Center._Y - _Center._Y));
        A = 1;
        B = 2 * _Center._X;
        C = square(_Center._X) + square(K) - 2 * K * _Center._Y
          + square(_Center._Y) - square(_Radius);
```

```
            intersect_pos._X = root_pos(A, B, C);
            intersect_pos._Y = K;
            intersect_neg._X = root_neg(A, B, C);
            intersect_neg._Y = K;

    }

    else
    {
            K =(( square(_Radius) - square(C2._Radius))
              + ( square(C2._Center._X) - square(_Center._X))
              + ( square(C2._Center._Y) - square(_Center._Y)))
              / ( 2 * (C2._Center._X - _Center._X));

            L = (_Center._Y - C2._Center._Y) / (C2._Center._X - _Center._X);
            A = 1 + L;
            B = 2 * ( K * L - L * _Center._X - _Center._Y);
            C = square(K) - (2 * K * _Center._X) + square(_Center._X)
              + square(_Center._Y) - ( 1 / square(_Kappa));

            intersect_pos._Y = root_pos(A, B, C);
            intersect_pos._X = K + intersect_pos._Y * L;
            intersect_neg._Y = root_neg(A, B, C);
            intersect_neg._X = K + intersect_neg._Y * L;

    // Determine Upstream and downstream Intersection Points
            A1 = atan2(intersect_pos._Y - _Center._Y, intersect_pos._X - _Center._X);
            A2 = atan2(intersect_neg._Y - _Center._Y, intersect_neg._X - _Center._X);
            A_ref = atan2(_Y - _Center._Y, _X - _Center._X);

            if ( _Kappa > 0.0 )
              {
                if ( positive_norm(A1 - A_ref) < positive_norm(A2 - A_ref) )
                        {
                          Intersect._X = intersect_pos._X;
                          Intersect._Y = intersect_pos._Y;
                        }
                else
                        {
                          Intersect._X = intersect_neg._X;
                          Intersect._Y = intersect_neg._Y;
                        }
              }
            else
              {
                if ( positive_norm(A1 - A_ref) > positive_norm(A2 - A_ref) )
                        {
                          Intersect._X = intersect_pos._X;
                          Intersect._Y = intersect_pos._Y;
                        }
                else
                        {
                          Intersect._X = intersect_neg._X;
                          Intersect._Y = intersect_neg._Y;
                        }
              }


            Intersect._Theta = C2._Theta;
            Intersect._Kappa = C2._Kappa;

       return Intersect;

    }
    return &intersect_pos;
}
```

```
Config
Circle :: image()
 {
   double gamma, x, y, t;

 // Calculate Angle of Vehicle from Center of Circle
   gamma = atan2(Robot._Y - _Center._Y, Robot._X - _Center._X);

 // Calculate X Position on Circle
   x = _Center._X + fabs(_Radius) * cos(gamma);

 // Calculate Y Position on Circle
   y = _Center._Y + fabs(_Radius) * sin(gamma);

 // Calculate Angle on Circle at X and Y
   t = norm(gamma + HPI * (_Kappa / fabs(_Kappa)));

 // Return Configuration of Image on Circle
   return Config(x, y, t);
 }


Config
Circle :: Project_image()
 {
   double gamma, x, y, t;

 // Calculate Angle of Vehicle from Center of Circle
   gamma = atan2(Projection._Y - _Center._Y, Projection._X - _Center._X);

 // Calculate X Position on Circle
   x = _Center._X + fabs(_Radius) * cos(gamma);

 // Calculate Y Position on Circle
   y = _Center._Y + fabs(_Radius) * sin(gamma);

 // Calculate Angle on Circle at X and Y
   t = norm(gamma + HPI * (_Kappa / fabs(_Kappa)));

 // Return Configuration of Image on Circle
   return Config(x, y, t);
 }

void
Circle :: Project_Start_image(Config &Intersect, double Mult)
 {
   double Phi, Gamma, x, y, t;

 // Calculate Angle of Intersect from Center of Circle
   Phi = atan2(Intersect._Y - _Center._Y, Intersect._X - _Center._X);

 // Calculate Angle of Projection from Intersect by Mult * s0
   Gamma = Phi - ( Mult * Projection.s0() / _Radius );

 // Calculate Projection X Position on Circle
   Projection._X = _Center._X + fabs(_Radius) * cos(Gamma) * (_Kappa/fabs(_Kappa));

 // Calculate Projection Y Position on Circle
   Projection._Y = _Center._Y + fabs(_Radius) * sin(Gamma) * (_Kappa/fabs(_Kappa));

 // Calculate Projection Angle on Circle at X and Y
   Projection._Theta = norm(Gamma + HPI * (_Kappa / fabs(_Kappa)));
 }
```

```
#ifndef Parabola_H
#define Parabola_H

#include "config.h"
#include "vehicle.h"
#include "line.h"
#include "util.h"
#include <math.h>

class Parabola : public Config
  {
    protected:

    // Structure
      Coordinate _Focus;

    public:

    // Constructors
      Parabola();
      Parabola(double, double, double, double, double);
      Parabola(Coordinate&, double, Coordinate&);
      Parabola(Line&, Coordinate&);
      Parabola(const Parabola&);

    // Destructor
      virtual ~Parabola() { }

    // Operators
      Parabola& operator=(const Parabola&);

    // Inline Mutators
      void set_Directive_x(double x) { _X = x; }
      void set_Directive_y(double y) { _Y = y; }
      void set_Directive_t(double t) { _Theta = t; }
      void set_Focus_x(double x)     { _Focus._X = x; }
      void set_Focus_y(double y)     { _Focus._Y = y; }


    // Inline Accessors
      Line Directrix()  const { return Line(_X, _Y, _Theta); }
      Coordinate Focus() const { return _Focus; }

    // Utilities
      Config intersects(Line&);

      Config image();
      Config Project_image();
      void Project_Start_image(Config&, double);

      double Close_Dist(Vehicle&, double, double);
  };

#endif
```

```
#include "parabola.h"
#include <math.h>

extern Vehicle Robot;
extern Vehicle Projection;

// Constructors
Parabola :: Parabola()
  : Config()
  {
    _Focus = Coordinate();
  }

Parabola :: Parabola(double dx, double dy, double dt, double fx, double fy)
  : Config(dx, dy, dt, 0)
  {
    _Focus = Coordinate(fx, fy);
  }

Parabola :: Parabola(Coordinate &d, double t, Coordinate &f)
  : Config(d._X, d._Y, t, 0)
  {
    _Focus = Coordinate(f._X, f._Y);
  }

Parabola :: Parabola(Line &l, Coordinate &p)
  : Config(l._X, l._Y, l._Theta)
  {
    _Focus = p;
  }

Parabola :: Parabola(const Parabola &p)
  : Config(p._X, p._Y, p._Theta)
  {
    _Focus = p._Focus;
  }

// Operators
Parabola&
Parabola :: operator=(const Parabola &P)
  {
    _X     = P._X;
    _Y     = P._Y;
    _Theta = P._Theta;
    _Kappa = P._Kappa;
    _Focus = P._Focus;
    return *this;
  }

Config
Parabola :: intersects(Line &L)
  {
    double A, B, C;
    double Dist1, Dist2, I1_Dist, I2_Dist;
    Config Intersect;
    Coordinate I1, I2;

    A = (square(sin(L._Theta)) * square(cos(_Theta)))
      + (square(sin(_Theta)) * square(cos(L._Theta)))
      - (2.0 * sin(L._Theta) * cos(L._Theta) * sin(_Theta) * cos(_Theta))
      - 1.0;

    B = (2.0
      * (((_X - L._X) * sin(_Theta)
      * ((sin(_Theta) * cos(L._Theta)) - (cos(_Theta) * sin(L._Theta))) )
      + ((_Y - L._Y) * cos(_Theta)
      * ((cos(_Theta) * sin(L._Theta)) - (sin(_Theta) * cos(L._Theta))) )
      + (_Focus._X - L._X) * cos(L._Theta)
```

```
    + (_Focus._Y - L._Y) * sin(L._Theta)));

  C = ( square(L._Y - _Y) * square(cos(_Theta)))
    + ( square(L._X - _X) * square(sin(_Theta)))
    + ( 2.0 * cos(_Theta) * sin(_Theta)
    * ( L._X * _Y + _Y * L._X - L._X * L._Y - _X * _Y))
    - square(L._X - _Focus._X)
    - square(L._Y - _Focus._Y);


  if ( A == 0 )
    {
          if ( B == 0)
            {
              Intersect._X = L._X + C * cos(L._Theta);
              Intersect._Y = L._Y + C * sin(L._Theta);
            }
          else
            {
        Dist1 =  -C / B;
              Intersect._X = L._X + Dist1 * cos(L._Theta);
              Intersect._Y = L._Y + Dist1 * sin(L._Theta);
            }
    }
  else
    {
          Dist1 = root_pos(A,B,C);
          Dist2 = root_neg(A,B,C);
          I1._X = L._X + Dist1 * cos(L._Theta);
          I1._Y = L._Y + Dist1 * sin(L._Theta);
          I2._X = L._X + Dist2 * cos(L._Theta);
          I2._Y = L._Y + Dist2 * sin(L._Theta);

          I1_Dist = ( I1._X - L._X ) * cos(L._Theta)
                  + ( I1._Y - L._Y ) * sin(L._Theta);

          I2_Dist = ( I2._X - L._X ) * cos(L._Theta)
                  + ( I2._Y - L._Y ) * sin(L._Theta);

          if ( I1_Dist < I2_Dist )
            {
              Intersect._X = I2._X;
              Intersect._Y = I2._Y;
            }
          else
            {
              Intersect._X = I1._X;
              Intersect._Y = I1._Y;
            }
    }

  return Intersect;
}

double
Parabola :: Close_Dist(Vehicle &Robot, double Length, double Phi)
{
  double Temp1, Temp2;

  if ( Length < 0 )
    {
    // j
          Temp1 = (Length * sin(norm(_Theta - Phi))) / (Length + cos(Phi));
          Temp2 = (Length * cos(norm(_Theta - Phi))) / (Length + cos(Phi));

          return ( square(Robot._X - _Focus._X - Temp1)
               + square(Robot._Y - _Focus._Y + Temp2));
    }
```

```
    else
      {
            Temp1 = (Length * sin(norm(_Theta + Phi))) / (Length + cos(Phi));
            Temp2 = (Length * cos(norm(_Theta + Phi))) / (Length + cos(Phi));

            return ( square(Robot._X - _Focus._X - Temp1)
                 + square(Robot._Y - _Focus._Y + Temp2));
      }
  }

Config
Parabola :: image()
  {
    double Length;
    double Phi, Phi_Lower, Phi_Upper;
    double Min1, Min2;
    double Upper = DPI;
    double Lower = 0.0;
    Config Image;

    Length = (_Focus._Y - _Y) * cos(_Theta) - (_Focus._X - _X) * sin(_Theta);

    for (int i = 0; i < 20; i++)
      {
            Phi_Lower = Lower + ( (Upper - Lower)/3.0);
       Phi_Upper = Upper - ( (Upper - Lower)/3.0);

            Min1 = this->Close_Dist(Robot, Length, Phi_Lower);
            Min2 = this->Close_Dist(Robot, Length, Phi_Upper);


            if ( Min1 > Min2 )
              {
                Lower = Phi_Lower;
              }
            else
              {
                Upper = Phi_Upper;
              }
      }

    if ( Min1 > Min2 )
      {
            Phi = norm(Phi_Upper);
      }
    else
      {
            Phi = norm(Phi_Lower);
      }

    if ( Length < 0.0 )
      {
            Image._X = _Focus._X + (Length * sin(norm(_Theta - Phi)) / (Length + cos(Phi)));
            Image._Y = _Focus._Y - (Length * cos(norm(_Theta - Phi)) / (Length + cos(Phi)));
            Image._Theta = norm(-Phi / 2.0 + _Theta);
            Image._Kappa = (1.0 / Length) * cube(cos(Phi/2.0));
      }
    else
      {
            Image._X = _Focus._X + (Length * sin(norm(_Theta + Phi)) / (Length + cos(Phi)));
            Image._Y = _Focus._Y - (Length * cos(norm(_Theta + Phi)) / (Length + cos(Phi)));
            Image._Theta = norm(Phi / 2.0 + _Theta);
            Image._Kappa = (1.0 / Length) * cube(cos(Phi/2.0));
      }

    return Image;
  }
```

```
Config
Parabola :: Project_image()
 {
   double Length;
   double Phi, Phi_Lower, Phi_Upper;
   double Min1, Min2;
   double Upper = DPI;
   double Lower = 0.0;
   Config Image;

   Length = (_Focus._Y - _Y) * cos(_Theta) - (_Focus._X - _X) * sin(_Theta);

   for (int i = 0; i < 20; i++)
    {
          Phi_Lower = Lower + ( (Upper - Lower)/3.0);
      Phi_Upper = Upper - ( (Upper - Lower)/3.0);

          Min1 = this->Close_Dist(Projection, Length, Phi_Lower);
          Min2 = this->Close_Dist(Projection, Length, Phi_Upper);


          if ( Min1 > Min2 )
           {
             Lower = Phi_Lower;
           }
          else
           {
             Upper = Phi_Upper;
           }
    }

   if ( Min1 > Min2 )
    {
          Phi = norm(Phi_Upper);
    }
   else
    {
          Phi = norm(Phi_Lower);
    }

   if ( Length < 0 )
    {
          Image._X = _Focus._X + (Length * sin(norm(_Theta - Phi)) / (Length + cos(Phi)));
          Image._Y = _Focus._Y - (Length * cos(norm(_Theta - Phi)) / (Length + cos(Phi)));
          Image._Theta = norm(-Phi / 2.0 + _Theta);
          Image._Kappa = (1.0 / Length) * cube(cos(Phi/2.0));
    }
   else
    {
          Image._X = _Focus._X + (Length * sin(norm(_Theta + Phi)) / (Length + cos(Phi)));
          Image._Y = _Focus._Y - (Length * cos(norm(_Theta + Phi)) / (Length + cos(Phi)));
          Image._Theta = norm(Phi / 2.0 + _Theta);
          Image._Kappa = (1.0 / Length) * cube(cos(Phi/2.0));
    }

   return Image;


 }


void
Parabola :: Project_Start_image(Config &Intersect, double Mult)
 {
   double Length;
   double Phi;
   double Gamma;
   Length = (_Focus._Y - _Y) * cos(_Theta) - (_Focus._X - _X) * sin(_Theta);
```

```
  Gamma  = 2 * Projection.s0() * Mult / Length;
  Phi    = acos( (fabs(Length) / sqrt(square(Intersect._X - _Focus._X)
             +                                    square(Intersect._Y - _Focus._Y))) - 1);

  if ( Length < 0 )
    {
          Phi += Gamma;
          Projection._X = _Focus._X + (Length * sin(norm(_Theta - Phi)) / (Length + cos(Phi)));
          Projection._Y = _Focus._Y - (Length * cos(norm(_Theta - Phi)) / (Length + cos(Phi)));
          Projection._Theta = norm(-Phi / 2.0 + _Theta);
          Projection._Kappa = (1.0 / Length) * cube(cos(Phi/2.0));
    }
  else
    {
          Phi -= Gamma;
          Projection._X = _Focus._X + (Length * sin(norm(_Theta + Phi)) / (Length + cos(Phi)));
          Projection._Y = _Focus._Y - (Length * cos(norm(_Theta + Phi)) / (Length + cos(Phi)));
          Projection._Theta = norm(Phi / 2.0 + _Theta);
          Projection._Kappa = (1.0 / Length) * cube(cos(Phi/2.0));
    }

}
```

```
#ifndef Cubic_H
#define Cubic_H

#include "config.h"


class Cubic : public Config
  {
    protected:

    // Structure
      double A_value;
      double L_value;

    public:

    // Constructors
      Cubic();
      Cubic(double, double, double);
      Cubic(const Config&);
      Cubic(const Config*);
      Cubic(const Cubic&);
      Cubic(const Cubic*);

    // Destructor
      virtual ~Cubic() { }

    // Operators
      Cubic& operator=(const Cubic&);
      friend ostream &operator<<(ostream&, Cubic&);

    // Inline Mutators
      void set_A(double a) { A_value = a; }
      void set_L(double l) { L_value = l; }

    // Inline Accessor
      double _A() const { return A_value; }
      double _L() const { return L_value; }

    // Utilities
      Config image();

      friend void   Solve1(Cubic&, Cubic&);
      friend Cubic  Split(Cubic&, Cubic&);
      friend double Cost(Cubic&, Cubic&, double, double, double, double);
  };

#endif
```

```
#include "cubic.h"
#include "util.h"
#include <math.h>

// Contructors
Cubic :: Cubic()
  : Config()
  {
    A_value = 0;
    L_value = 0;
  }


Cubic :: Cubic(double x, double y, double t)
  : Config(x, y, t)
  {
    A_value = 0;
    L_value = 0;
  }


Cubic :: Cubic(const Config &c)
  : Config(c._X, c._Y, c._Theta)
  {
    A_value = 0;
    L_value = 0;
  }


Cubic :: Cubic(const Config *c)
  : Config(c->_X, c->_Y, c->_Theta)
  {
    A_value = 0;
    L_value = 0;
  }

Cubic :: Cubic(const Cubic &c)
  : Config(c._X, c._Y, c._Theta)
  {
    A_value = c.A_value;
    L_value = c.L_value;
  }


Cubic :: Cubic(const Cubic *c)
  : Config(c->_X, c->_Y, c->_Theta)
  {
    A_value = c->A_value;
    L_value = c->L_value;
  }


// Operators
Cubic&
Cubic :: operator=(const Cubic &C)
  {
    _X      = C._X;
    _Y      = C._Y;
    _Theta  = C._Theta;
    _Kappa  = C._Kappa;
    A_value = C.A_value;
    L_value = C.L_value;
    return *this;
  }
```

```
ostream &operator<<(ostream &strm, Cubic &p)
 {
   return strm << "( "  << p._X
                       << ", " << p._Y
                       << ", " << p._Theta
                       << " ) " << p._A()
                       << ", " << p._L();
 }


// Utilities
Config
Cubic :: image()
 {
   cout << "Entered Cubic image \n" << flush;

 }


void Solve1(Cubic &C, Cubic &G)
 {
 //
   double Alpha;

 // Calculate Theta Between Two Coordinates
   Alpha    = norm( atan2(G._Y - C._Y, G._X - C._X) - C._Theta) * 2;

 // Set Cubic Alph Vale to Calculated Alpha
   C.A_value = Alpha;

 // Calculate Dist Between Two coordinates
   C.L_value = compute_dist(G._X, G._Y, C._X, C._Y) / lookup(fabs(Alpha));
 }


Cubic Split(Cubic &C, Cubic &G)
 {
   double co, xc, yc, r, xm, ym, tm, g1, g2, g, w;
   double alpha, alpha1, beta1, eta1, eta2, w1;
   double costg, costg1, costg2;

   alpha  = norm(G._Theta - C._Theta);
   alpha1 = fabs(alpha);

   if ( alpha1 < ZERA)
    {
            xm = ( C._X + G._X ) / 2;
            ym = ( C._Y + G._Y ) / 2;
    }
   else
    {
            co = 1.0 / tan( alpha / 2.0 );
            xc = ( C._X + G._X + co * (C._Y - G._Y)) / 2.0;
            yc = ( C._Y + G._Y + co * (G._X - C._X)) / 2.0;
            r  = sqrt( square(C._X - xc) + square(C._Y - yc));

            if (alpha > 0.0)
             {
               g1   = atan2(C._Y - yc, C._X - xc);
               g2   = atan2(G._Y - yc, G._X - xc);
               eta1 = C._Theta - HPI;
               eta2 = G._Theta - HPI;
             }
            else
             {
               g1   = atan2(G._Y - yc, G._X - xc);
               g2   = atan2(C._Y - yc, C._X - xc);
               eta1 = G._Theta + HPI;
```

```
            eta2 = C._Theta + HPI;
          }

        g = ( g1 + g2 ) / 2.0;
        w = alpha1 / 2.0;

        w1 = positive_norm(eta1 - g1);
        if ( (w1 < alpha1) && (2.0 * w1 < alpha1) )
          {
            g = eta1;
            w = w1;
          }

        w1 = positive_norm(g2 - eta2);
        if ( (w1 < alpha1) && ( 2.0 * w1 < alpha1) )
          {
            g = eta2;
            w = w1;
          }

        costg = Cost(C, G, xc, yc, r, g);

        while ( w > ZERA )
          {
            w = w / 2.0;
            costg1 = Cost(C, G, xc, yc, r, g + w);
            if ( costg1 < costg )
                    {
                      g = g + w;
                      costg = costg1;
                    }
            else
                    {
                      costg2 = Cost(C, G, xc, yc, r, g - w);
                      if ( costg2 < costg )
                        {
                                g = g - w;
                                costg = costg2;
                        }
                    }
          }
        xm = xc + r * cos(g);
        ym = yc + r * sin(g);
    }
   beta1 = atan2(ym - C._Y, xm - C._X);
   tm    = beta1 + norm(beta1 - C._Theta);
   return Cubic(xm, ym, tm);
}


double Cost(Cubic &C, Cubic &G, double xc, double yc, double r, double g)
 {
   double x, y, dist1, dist2, alpha1, alpha2;

   x = xc + r * cos(g);
   y = yc + r * sin(g);
   dist1  = compute_dist(C._X, x, C._Y, y);
   alpha1 = 2.0 * norm(atan2(y - C._Y, x - C._X) - C._Theta);

   dist2  = compute_dist(G._X, x, G._Y, y);
   alpha2 = 2.0 * norm(atan2(G._Y - y, G._X - x) - G._Theta);

   return ( costf(alpha1, dist1) + costf(alpha2, dist2) );
 }
```

APPENDIX B: C++ CODE FOR INTERPRETER MODULE

```
#include "list.h"
#include "mml.h"
#include "config.h"
#include "vehicle.h"
#include "line.h"
#include "circle.h"
#include "cubic.h"
#include "sonar.h"
#include "segment.h"
#include "util.h"
#include <fstream.h>

List Buffer;
List Wait_Buffer;

extern Vehicle Robot;
extern Vehicle Projection;
Config *IPath;
extern int need_intersect;

extern double PDist;

Object Current_Parameter;
Object Current_Instructon;
Config PImage;
Config Intersect;
Config *Current_PPath;

int seq_status;
// int Need_Further_Processing;
int Need_Config = 0;

int TRANSITION = 0;
int Path_to_intersect = NONE;

int Close_Enough = 0;

int Service_Flag = 0;


// Function
  void set_error(int code)
    {
//          Instruction temp   = Instruction(ERROR, code)
            Buffer = Buffer.Push( Object(ERROR, code));
    }

  void solve(Cubic &Curr, Cubic &Goal)
    {
            double  Beta;
      Cubic   New_Goal;

    // Check for Legal Pair of Configurations
            if ( (zera(Curr._Y - Goal._Y)) && (zera(Curr._X - Goal._X)) )
              set_error(ECODE2);

    // Calculate Angle between the Two Configurations
            Beta = atan2(( Goal._Y - Curr._Y ), ( Goal._X - Curr._X ));

    // Check for Symmetric Configurations
            if (fabs(norm(Goal._Theta - Beta) - norm(Beta - Curr._Theta)) < ZERA)
              {
              // Solve for Curve Parameters
                Solve1(Curr, Goal);
```

```
                Buffer = Buffer.Push( Object(MOVEMENT, CUBIC_SPIRAL, 1, Curr));

              }
            else
              {
                New_Goal = Split(Curr, Goal);
                Solve1(Curr, New_Goal);
                 Buffer = Buffer.Push( Object(MOVEMENT, CUBIC_SPIRAL, 1, Curr));
                Solve1(New_Goal, Goal);
            Buffer = Buffer.Push( Object(MOVEMENT, CUBIC_SPIRAL, 1, New_Goal));
              }
    }

double Find_S0_Window(double Mult)
 {
   double Total_Dist = 0;
   double Prev_Dist  = 0;
   double Done = 0;

 // Set Close Enough flag to NO
   Close_Enough = NO;

 // Loop Until S0 Doesn't Cross
   while (! Done)
    {
    // Calculate PImage
            PImage  = Current_PPath->Project_image();

    // Calculate Total Dist Traveled
            Total_Dist += Project_Update_config(Current_PPath, PImage);
    // Check if Projection Crosses Over New Path By Sign Change
      if ( Prev_Dist * PDist < 0 )
              {
              // Increase Mult By 1
                Mult++;
        // Reset Image at New Transition Point
                IPath->Project_Start_image(Intersect, Mult);

        // Reset Variables to 0;
                Total_Dist = PDist = Prev_Dist = 0;
              }

    // Check if Projection is Close Enough
            if ( PImage.Project_Transition(Current_PPath))
              {
        // No Further Processing Required
                Close_Enough = 1;

        // Return Muliplier for S0
                return Mult;
              }

    // Check if Projection Never Crossed
            if ( Total_Dist > 6.0 * Projection.s0())
              {
              // Return Muliplier for S0
          return Mult;
              }

    // Set Prev Dist to Dist to Check for Sign Change
            Prev_Dist = PDist;

   // End While
          }
   return Mult;

// End Function
  }
```

```
int Test_S0()
  {
    double Total_Dist = 0;
    double Prev_Dist  = 0;
    double Done = 0;

  // Set Close Enough and Over Shot flags to NO
    Close_Enough = NO;

  // Loop Until Transistion Point Doesn't Cross
    while (! Done)
      {
      // Calculate Image
              PImage  = Current_PPath->Project_image();

      // Calculate Total Dist Traveled
              Total_Dist += Project_Update_config(Current_PPath, PImage);

      // Check if Projection Crosses Over New Path By Sign Change
        if ( Prev_Dist * PDist < 0 )
                {
                // Set OverShoot Flag to Yes
            return OVER_SHOT;
                }

      // Check if Projection is Close Enough
              if ( PImage.Project_Transition(Current_PPath))
                {
          // Return Close Enough Flag
                  return CLOSE_ENOUGH;
                }
      // Check if Projection Never Crossed
              if ( Total_Dist > 6.0 * Projection.s0())
                {
          // Return Didn't Cross Flag
                  return DID_NOT_CROSS;
                }

      // Set Prev Dist to Dist to Check for Sign Change
              Prev_Dist = PDist;

    // End While
            }

// End Function
  }




// Function Get Transition
  double Get_Transition()
    {
            double Mult  = 1;
      double Upper, Lower;
            int    S0_Tracer;

    // Set Image on Path at S0
            IPath->Project_Start_image(Intersect, Mult);

    // Find (Multiplier * S0) that Doesn't Cross New Path
            Upper = Find_S0_Window(Mult);

    // Set Lower Bound to 1 - Mult
            Lower = Upper - 1.0;
```

```
// If Projection isn't Close Enough then Break Down Further
        if ( Close_Enough )
          {
            return Mult;
          }
        else
    {
    // Loop
            for (int i = 0; i <= 4; i++)
             {
      // Set Mult to Middle of Widow
               Mult  = (Upper + Lower) / 2;

             // Set Image on Path at S0
                    IPath->Project_Start_image(Intersect, Mult);

             // Check if New S0 Crosses the New Path
         S0_Tracer = Test_S0();

             //
                    switch ( S0_Tracer )
                     {
        // Check What Projection Did
                        case OVER_SHOT:
           // Increase Lower Bound by Half
                            Lower = (Upper + Lower) / 2;
             break;

                        case DID_NOT_CROSS:
           // Decrease Upper Bound by Half
                            Upper = (Upper + Lower) / 2;
             break;

                        case CLOSE_ENOUGH:
                           return Mult;
                                break;

                        default:
                           break;
                      }
             }
          }

        if ( S0_Tracer == OVER_SHOT )
          return Upper;
        else
          return Mult;
  }


void Push_Parameter_On_Buffer(int Parameter, double Value)
 {
  // Determine if Command is Placed in Buffer or Wait Buffer
        switch ( Path_to_intersect)
    {
          // Push on Wait Buffer until Intersection Found
            case FORWARD_LINE:
            case FORWARD_CIRCLE:
            case LINE:
      case CIRCLE:
      case PARABOLA:
                    Wait_Buffer = Wait_Buffer.Push( Object(PARAMETER, Parameter, Value));
                    break;

    // Push on Buffer
            case BACKWARD_LINE:
            case BACKWARD_CIRCLE:
```

```
                case NONE:
                case CUBIC:
                        Buffer = Buffer.Push( Object(PARAMETER, Parameter. Value));
                        break;

                default:
            break;
                }
    }
void Push_Stationary_On_Buffer(int Class)
  {
    // Determine if Command is Placed in Buffer or Wait Buffer
            switch ( Path_to_intersect)
       {
              // Push on Wait Buffer until Intersection Found
                case FORWARD_LINE:
                case FORWARD_CIRCLE:
                case LINE:
          case CIRCLE:
          case PARABOLA:
                        Wait_Buffer = Wait_Buffer.Push( Object(STATIONARY, Class));
                        break;

        // Push on Buffer
                case BACKWARD_LINE:
                case BACKWARD_CIRCLE:
                case NONE:
                case CUBIC:
                        Buffer = Buffer.Push( Object(STATIONARY, Class));
                        break;

                default:
            break;
                }
    }

void Push_Two_Parameter_Object_On_Buffer(int Parm, double V1, double V2)
  {
    // Determine if Command is Placed in Buffer or Wait Buffer
            switch ( Path_to_intersect)
       {
              // Push on Buffer
                case FORWARD_LINE:
                case FORWARD_CIRCLE:
                case LINE:
          case CIRCLE:
          case PARABOLA:
                case BACKWARD_LINE:
                case BACKWARD_CIRCLE:
                case NONE:
                case CUBIC:
                        Buffer = Buffer.Push( Object(PARAMETER, Parm, V1, V2));
                        break;

                default:
            break;
                }
    }

/*
  void mark_critical()
    {
            wsyn_q = 1;
    }


  void unmark_critical()
    {
```

```
            wsyn_q = 1;
            while (!wsyn_q = 0 );
    }

// Function Set Acceleration
  void Set_Acel(double s)
    {
            Accel = s
    }
*/
////////////////////////////////////////////////////////////////////////

// Function Set S0
  void Set_S0(double s)
    {
    // Push Command on Instruction Buffer in Appropriate Sequential Order
            Push_Parameter_On_Buffer( SET_S0, s);

    // Set Projection S0
            Projection.set_S0(s);
    }

// Fuction Reset S0
  void Reset_S0(double s)
    {
    // Set S0 Imediately
            Robot.set_S0(s);

    }

////////////////////////////////////////////////////////////////////////
// Function Set Speed
  void Set_Speed(int s)
    {
    // Push Command on Instruction Buffer in Appropriate Sequential Order
            Push_Parameter_On_Buffer( SET_SPEED, s);

    // Set Projection Speed
      Projection.set_Speed(s);
    }

// Function
  void Reset_Speed(int s)
    {
    // Change Speed Imediately
            Robot.set_Speed(s);
    }

////////////////////////////////////////////////////////////////////////
// Function Set Linear Acceleration
  void Set_Accel(double a)
    {
    // Push Command on Instruction Buffer in Appropriate Seqential Order
            Push_Parameter_On_Buffer( SET_LACCEL, a);
    }

// Function ReSet Linear Acceleration
  void Reset_Accel(double a)
    {
    // Reset Linear Acceletation Immediately
            Robot.set_L_Accel(a);
    }


////////////////////////////////////////////////////////////////////////
// Function Reset Robot Configuration
  void Reset_Rob(double x, double y, double t)
    {
```

```
    // Reset Robot's x,y and theta Immediately
            Robot.set_x(x);
            Robot.set_y(y);
            Robot.set_theta(t);
    }

  void Reset_Rob(Config &c)
    {
    // Reset Robot's x,y and theta Immediately
            Robot.set_x(c._X);
            Robot.set_y(c._Y);
            Robot.set_theta(c._Theta);
    }

  void Reset_Rob(Vehicle &v)
    {
    // Reset Robot's x,y and theta Immediately
            Robot.set_x(v._X);
            Robot.set_y(v._Y);
            Robot.set_theta(v._Theta);
    }


////////////////////////////////////////////////////////////////////////
// Function Get Robot Configuration
  void Get_Rob(Config &c)
    {
            c.set_x(Robot._X);
            c.set_y(Robot._Y);
            c.set_theta(Robot._Theta);
            c.set_kappa(Robot._Kappa);
    }

  void Get_Rob(Vehicle &v)
    {
            v = Robot;
    }


////////////////////////////////////////////////////////////////////////
// Function Get Current Buffer Object
  void Get_Buf(Object &I)
    {
            I = Buffer.Top();
    }



////////////////////////////////////////////////////////////////////////
// Function Trace Robot
  void Trace_Robot()
    {
    // Push Command on Instruction Buffer in Appropriate Sequential Order
            Push_Parameter_On_Buffer( TRACE_ROBOT, FILLER );
    }

  void Trace_Sim()
    {
    // Push Command on Instruction Buffer in Appropriate Sequential Order
            Push_Parameter_On_Buffer( TRACE_SIMULATOR, FILLER );
    }


// Function Enable Sonar Group
  void Enable_Sonar_Group(int s)
    {
    // Push Command on Instruction Buffer in Appropriate Sequential Order
            Push_Parameter_On_Buffer( ENABLE_SONAR, s);
```

```
    }

// Function Disable Sonar Group
  void Disable_Sonar_Group(int s)
    {
    // Push Command on Instruction Buffer in Appropriate Sequential Order
            Push_Parameter_On_Buffer( DISABLE_SONAR, s);

    }

// Function Get Specific Sonar Range
  double Get_Sonar_Range(int s)
    {
    // Return Specific Sonar Range
            return Execute_Get_Sonar_Range(s);

    }

// Function Get Updated Sonar Range
  double Wait_For_Updated_Sonar_Range(int s)
    {
      return Execute_Wait_For_Updated_Sonar_Range(s);
    }


// Function Get Return's Globol Position from Specific Sonar
  Config Get_Return_Position(int s)
    {
      return Execute_Get_Sonar_Range_Position(s);
    }


// Function
  void Wait_Until_LT_Sonar_Range(int Sonar_Num, double Limit)
    {
    // Push Command on Instruction Buffer in Appropriate Sequential Order
            Push_Two_Parameter_Object_On_Buffer( SONAR_RANGE_LT, Sonar_Num, Limit);
    }


// Function
  void Wait_Util_GT_Sonar_Range(int Sonar_Num, double Limit)
    {
    // Push Command on Instruction Buffer in Appropriate Sequential Order
            Push_Two_Parameter_Object_On_Buffer( SONAR_RANGE_GT, Sonar_Num, Limit);

    }

////////////////////////////////////////////////////////////////////////////
//
// Function
  void Set_robot(Config &c)
    {
    // Check if robot is Moving
            if (! seq_status == SSTOP)
        {
        // Flag Error Message: Robot is Moving
                set_error(ECODE2);
        }
                else
                  {
        // Load Set Robot Command into Buffer
                  Buffer = Buffer.Push(Object(STATIONARY, SET_ROBOT, c));
                  }
    }
```

```
// Function
  void Set_robot(double x, double y, double t)
    {
            Set_robot(Config(x, y, t, 0));
    }

////////////////////////////////////////////////////////////////////////
// Function Stop Robot
  void Stop()
    {
            Robot.set_Speed(0);
    }

// Function
  void Stop_robot()
    {
    // Push Command on Instruction Buffer at Appropriate Sequential Order
            Buffer = Buffer.Imediate_Push(Object(STATIONARY, STOP_ROBOT));

    // Set Last Path Status to Complete
            Buffer.Bottom2().set_Status(COMPLETE);
    }


////////////////////////////////////////////////////////////////////////
// Function Terminate Program
  void End()
    {
    // Push Command on Instruction Buffer in Appropriate Sequential Order
            Push_Stationary_On_Buffer(END);

    // Set Last Path Status to Complete
            Buffer.Bottom2().set_Status(COMPLETE);

    }


////////////////////////////////////////////////////////////////////////
// Functions Halt and Resume
  void Halt()
    {
    // Push Command on Instruction Buffer at Appropriate Sequential Order
            Buffer = Buffer.Imediate_Push(Object(STATIONARY, STOP_ROBOT));
    }

  void Resume()
    {
    // Set Last Path Status to Complete
            Buffer.Bottom().set_Status(COMPLETE);
    }


////////////////////////////////////////////////////////////////////////
// Function
  void Rotate(double theta)
    {
    // Convert Theta to Radians
            theta = theta * DEG_RAD;

    // Check if robot is Moving
            if (seq_status == MOVING)
              {
       // Flag Error Message: Robot is Moving
                set_error(ECODE2);
              }
            else
       {
       // Add Rotate Command to Instruction Buffer
```

```
        Buffer = Buffer.Push(Object(STATIONARY, ROTATE, theta));
      }
    }


  void Rotate_To(double theta)
    {
    // Convert Theta to Radians
            theta = theta * DEG_RAD;

    // Check if robot is Moving
            if (seq_status == MOVING)
              {
      // Flag Error Message: Robot is Moving
                set_error(ECODE2);
              }
            else
      {
      // Add Rotate Command to Instruction Buffer
        Buffer = Buffer.Push(Object(STATIONARY, ROTATE_TO, theta));
      }
    }



////////////////////////////////////////////////////////////////////////
// Funciton
  void Pop_Wait_Buffer()
    {
    // Loop until Buffer is Empty
            while (Wait_Buffer.last())
              {
      // Push Wait Buffer onto Buffer
              Buffer = Buffer.Push(Wait_Buffer.Pop());
      }
    }


// Function Forward Path
  void FPath(Line &Temp_Path)
    {
    // Check Legality of Command
            switch ( Path_to_intersect)
      {
             // Illegal Transition Forward Path to Forward Path
              case FORWARD_LINE:
              case FORWARD_CIRCLE:
        {
                      set_error(ECODE1);
                      break;
        }

      // Illegal Transition Path to Forward Path
              case LINE:
        case CIRCLE:
              case PARABOLA:
        {
                      set_error(ECODE2);
                      break;
        }

      // Legal Transition Backward Path to Forward Path
             case BACKWARD_LINE:
             case BACKWARD_CIRCLE:
        {
        // Solve Cubic Spiral
                      solve(IPath, Temp_Path);
```

```
        // Set Path to Forward Path
                IPath = &Temp_Path;

        // Push Path onto Buffer
          Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                break;
        }

      // Legal Transition Start with a Forward Path
            case NONE:
              {
        // Set Path to Forward Path
                IPath = &Temp_Path;

        // Push Path onto Buffer
          Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                break;
        }

      // Legal Transition: Posture to a Forward Line
        case FORWARD_CUBIC:
            case CUBIC:
            case BACKWARD_CUBIC:
              {
        // Solve Cubic Spiral
                solve(IPath, Temp_Path);

        // Set Path to Forward Path
                IPath = &Temp_Path;

        // Push Path onto Buffer
          Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                break;
        }

            default:
         break;
            }
    // Set Sequential Status to Moving
          seq_status = MOVING;

    // Set Path to Intersect to a Forward Line
          Path_to_intersect = FORWARD_LINE;
    }


// Function
  void FPath(Circle &Temp_Path)
    {
    // Check Legality of Command
          switch ( Path_to_intersect)
      {
            // Illegal Transition Forward Path to Forward Path
              case FORWARD_LINE:
              case FORWARD_CIRCLE:
        {
                set_error(ECODE1);
                break;
        }

      // Illegal Transition Path to Forward Path
            case LINE:
        case CIRCLE:
            case PARABOLA:
        {
                set_error(ECODE2);
                break;
```

```
        }

      // Legal Transition Backward Path to Forward Path
            case BACKWARD_LINE:
            case BACKWARD_CIRCLE:
        {
        // Solve Cubic Spiral
                    solve(IPath, Temp_Path);

        // Set Path to Forward Path
                    IPath = &Temp_Path;

        // Push Path onto Buffer
          Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                break;
        }

      // Legal Transition Start with a Forward Path
            case NONE:
              {
        // Set Path to Forward Path
                    IPath = &Temp_Path;

        // Push Path onto Buffer
          Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                    break;
        }

      // Legal Transition: Posture to a Forward Line
        case FORWARD_CUBIC:
            case CUBIC:
            case BACKWARD_CUBIC:
              {
        // Solve Cubic Spiral
                    solve(IPath, Temp_Path);

        // Set Path to Forward Path
                    IPath = &Temp_Path;

        // Push Path onto Buffer
          Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                    break;
        }

              default:
         break;
            }
    // Set Sequential Status to Moving
            seq_status = MOVING;


    // Set Path to Intersect to a Forward Line

            Path_to_intersect = FORWARD_CIRCLE;
    }


// Function
  void FPath(double x, double y, double t)
    {
            FPath(Line(x, y, t, 0));
    }


// Function
  void FPath(double x, double y, double t, double k)
    {
            if ( k == 0 )
```

```
            FPath(Line(x, y, t, 0));
          else
            FPath(Circle(x, y, t, k));
    }



// Function
  void Path(Line &Temp_Path)
    {
      double Mult;
    // Check Legality of Command
          switch ( Path_to_intersect)
       {
            // Legal Transition Forward Path or Path to a Path
       // Intersect Needed
            case FORWARD_LINE:
            case FORWARD_CIRCLE:
         case FORWARD_PARABOLA:
            case LINE:
         case CIRCLE:
            case PARABOLA:
              {
          // Calculate Intersect
                  Intersect = IPath->intersects(Temp_Path);

            // Set Path to Temp Path
                  Current_PPath = &Temp_Path;

            // Calculate Transition Point
                  Mult = Get_Transition();

            // Push Intersect Instruction onto Buffer
                  Buffer = Buffer.Push(Object(MOVEMENT, INTERSECT, 1, Intersect, Mult));

            // Set Last Path Status to Complete
                  Buffer.Bottom2().set_Status(COMPLETE);

         // Push Wait Buffer onto Buffer
                  Pop_Wait_Buffer();

            // Set Path to Temp Path
                  IPath = &Temp_Path;

            // Push New Path onto Buffer with status of Incomplete
              Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                  break;
              }

          // Legal Transition: Backward Path to Path.
       // No Intersect Needed
            case BACKWARD_LINE:
            case BACKWARD_CIRCLE:
            case BACKWARD_PARABOLA:
            case NONE:
              {
         // Set Path to New Path
                  IPath   = &Temp_Path;

         // Push New Path onto Buffer with status of Incomplete
              Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
              break;
         }

       // Illegal Transition: Path to Cubic Spiral Config
         case FORWARD_CUBIC:
```

```
                case CUBIC:
                case BACKWARD_CUBIC:
        {
                    set_error(ECODE2);
                        break;
        }

      // Default: Unkown New Path
                default:
          break;

     // End Switch
                }

   // Set Sequential Status to Moving
            seq_status = MOVING;

   // Set Path to Intersect to Line
            Path_to_intersect = LINE;
   }


// Function
  void Path(Circle &Temp_Path)
   {
            double Mult;
   // Check Legality of Command
            switch ( Path_to_intersect)
      {
              // Legal Transition Forward Path or Path to a Path
       // Intersect Needed
                case FORWARD_LINE:
                case FORWARD_CIRCLE:
        case FORWARD_PARABOLA:
                case LINE:
        case CIRCLE:
                case PARABOLA:
                 {
         // Calculate Intersect
                        Intersect = IPath->intersects(Temp_Path);

                  // Set Path to Temp Path
                        Current_PPath = &Temp_Path;

                  // Calculate Transition Point
                        Mult = Get_Transition();

         // Push Intersect Instruction onto Buffer
                        Buffer = Buffer.Push(Object(MOVEMENT, INTERSECT, 1, Intersect, Mult));

         // Set Last Path Status to Complete
                        Buffer.Bottom2().set_Status(COMPLETE);

         // Push Wait Buffer onto Buffer
                        Pop_Wait_Buffer();

                  // Set Path to Temp Path
                        IPath = &Temp_Path;

                  // Push New Path onto Buffer with status of Incomplete
                    Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                        break;
                }


              // Legal Transition: Backward Path to Path.
       // No Intersect Needed
                case BACKWARD_LINE:
```

```
                case BACKWARD_CIRCLE:
                case BACKWARD_PARABOLA:
                case NONE:
                  {
          // Set Path to New Path
                        IPath  = &Temp_Path;

          // Push New Path onto Buffer with status of Incomplete
                        Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                        break;
          }

        // Illegal Transition: Path to Cubic Spiral Config
          case FORWARD_CUBIC:
                case CUBIC:
                case BACKWARD_CUBIC:
          {
                        set_error(ECODE2);
                              break;
          }

        // Default: Unkown New Path
                default:
            break;

      // End Switch
                }

    // Set Sequential Status to Moving
              seq_status = MOVING;

    // Set Path to Intersect to Circle
              Path_to_intersect = CIRCLE;
    }


// Function
  void Path(double x, double y, double t)
    {
              Path(Line(x,y,t,0));
    }


// Function
  void Path(double x, double y, double t, double k)
    {
      if ( k == 0)
                Path(Line(x,y,t,0));
              else
                Path(Circle(x,y,t,k));

    }


  void Path(Parabola &Temp_Path)
    {
              double Mult;

    // Check Legality of Command
              switch ( Path_to_intersect)
        {
                 // Legal Transition: Forward Line or Line to Parabola
                   case FORWARD_LINE:
                   case LINE:
                     {
            // Calculate Intersect
                              Intersect = IPath->intersects(Temp_Path);
```

```
                // Set Path to Temp Path
                        Current_PPath = &Temp_Path;

                // Calculate Transition Point
                        Mult = Get_Transition();

        // Push Intersect Instruction onto Buffer
                        Buffer = Buffer.Push(Object(MOVEMENT, INTERSECT, 1, Intersect, Mult));

        // Set Last Path Status to Complete
                        Buffer.Bottom2().set_Status(COMPLETE);

        // Push Wait Buffer onto Buffer
                        Pop_Wait_Buffer();

                // Set Path to Temp Path
                        IPath = &Temp_Path;

                // Push New Path onto Buffer with status of Incomplete
                  Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                        break;
                }

      // Illegal Transition Circle or Parabola to Parabola
              case FORWARD_CIRCLE:
        case FORWARD_PARABOLA:
        case CIRCLE:
              case PARABOLA:
         {
                        set_error(ECODE2);
         }

      // Legal Transition
              case BACKWARD_LINE:
              case BACKWARD_CIRCLE:
              case BACKWARD_PARABOLA:
        case NONE:
                        IPath = &Temp_Path;
                  Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                  break;


        case FORWARD_CUBIC:
              case CUBIC:
        case BACKWARD_CUBIC:
                        set_error(ECODE2);
                        break;

              default:
          break;
             }

    // Set Sequential Status to Moving
            seq_status = MOVING;

    // Set Path to Intersect to Parabola
            Path_to_intersect = PARABOLA;
    }




// Function For a Parabola in the form of a Directix and Focus
  void Path(Line &l, Coordinate &c)
    {
      Path(Parabola(l, c));
```

```
    }


// Function For a Parabola in the form of Directix as x,y,t and Focus as x,y
  void Path(double dx, double dy, double dt, double fx, double fy)
    {
            Path(Parabola(dx, dy, dt, fx, fy));
    }


// Function
  void BPath(Line &Temp_Path)
    {
      double Mult;
    // Check Legality of Command
            switch ( Path_to_intersect)
       {
             // 1
               case FORWARD_LINE:
               case FORWARD_CIRCLE:
          case FORWARD_PARABOLA:
               case LINE:
          case CIRCLE:
               case PARABOLA:
                 {
           // Calculate Intersect
                        Intersect = IPath->intersects(Temp_Path);

                 // Set Path to Temp Path
                        Current_PPath = &Temp_Path;

                 // Calculate Transition Point
                        Mult = Get_Transition();

          // Push Intersect Instruction onto Buffer
                       Buffer = Buffer.Push(Object(MOVEMENT, INTERSECT, 1, Intersect, Mult));

          // Set Last Path Status to Complete
                        Buffer.Bottom2().set_Status(COMPLETE);

          // Push Wait Buffer onto Buffer
                        Pop_Wait_Buffer();

                 // Set Path to Temp Path
                        IPath = &Temp_Path;

                 // Push New Path onto Buffer with status of Incomplete
                   Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                        break;
                 }

               case BACKWARD_LINE:
               case BACKWARD_CIRCLE:
               case BACKWARD_PARABOLA:
          case NONE:
           {
                        IPath = &Temp_Path;
                 Buffer = Buffer.Push( Object(MOVEMENT, PARTIAL_PATH, 1, Temp_Path) );
                        break;
           }


          case FORWARD_CUBIC:
               case CUBIC:
             set_error(ECODE2);
                       break;

               default:
```

```
            break;
              }
             seq_status = SSTOP;
      Path_to_intersect = BACKWARD_LINE;
    }


// Function
  void BPath(Circle &Temp_Path)
    {
    // Check Legality of Command
            switch ( Path_to_intersect)
              {
         double Mult;
              // 1
                case FORWARD_LINE:
                case FORWARD_CIRCLE:
                case LINE:
          case CIRCLE:
                case PARABOLA:
                  {
          // Calculate Intersect
                        Intersect = IPath->intersects(Temp_Path);

                  // Set Path to Temp Path
                        Current_PPath = &Temp_Path;

                  // Calculate Transition Point
                        Mult = Get_Transition();

          // Push Intersect Instruction onto Buffer
                       Buffer = Buffer.Push(Object(MOVEMENT, INTERSECT, 1, Intersect, Mult));

          // Set Last Path Status to Complete
                        Buffer.Bottom2().set_Status(COMPLETE);

          // Push Wait Buffer onto Buffer
                        Pop_Wait_Buffer();

                  // Set Path to Temp Path
                        IPath = &Temp_Path;

                  // Push New Path onto Buffer with status of Incomplete
                    Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                        break;
                  }

                case BACKWARD_LINE:
                case BACKWARD_CIRCLE:
                case NONE:
          {
                        IPath = &Temp_Path;
                 Buffer = Buffer.Push( Object(MOVEMENT, PARTIAL_PATH, 1, Temp_Path) );
                        break;
          }


         case FORWARD_CUBIC:
                case CUBIC:
                       set_error(ECODE2);
                       break;

                default:
           break;
              }
            seq_status = SSTOP;
```

```
      Path_to_intersect = BACKWARD_CIRCLE;
    }

// Function
  void BPath(double x, double y, double t)
    {
            BPath(Line(x, y, t, 0));
    }


// Function
  void BPath(double x, double y, double t, double k)
    {
    // Check if Line or Circle
            if ( t == 0)
              BPath(Line(x, y, t, 0));
            else
              BPath(Circle(x, y, t, k));
    }

// Function
  void Posture(Cubic &Temp_Path)
    {

    // Check Legality of Command
            switch ( Path_to_intersect)
       {
              // 1
                case FORWARD_LINE:
                case FORWARD_CIRCLE:
                case LINE:
          case CIRCLE:
          case PARABOLA:
                        set_error(ECODE2);
                        break;


                case BACKWARD_LINE:
                case BACKWARD_CIRCLE:
                case BACKWARD_PARABOLA:
          {
                        solve(IPath, Temp_Path);
                        IPath = &Temp_Path;
//              Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                        break;
           }

          case NONE:
                        IPath = &Temp_Path;
//              Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                        break;


          case FORWARD_CUBIC:
                case CUBIC:
                case BACKWARD_CUBIC:
           {
                        solve(IPath, Temp_Path);
                        IPath = &Temp_Path;
              Buffer = Buffer.Push( Object(MOVEMENT, FULL_PATH, Temp_Path) );
                        break;
           }

                default:
           break;
              }
            seq_status = MOVING;
            Path_to_intersect = CUBIC;
```

```
    }


// Function
  void Posture(double x, double y, double t)
    {
      Posture( Cubic(x, y, t));
    }


  void Print_Buffer()
    {
      while (Buffer.last())
              cout<< Buffer.Pop() << "\n" << flush;
    }
```

APPENDIX C: C++ CODE FOR CONTROLER MODULE

```
#include "list.h"
#include "mml.h"
#include "vehicle.h"
#include "line.h"
#include "cubic.h"
#include "ping.h"
#include "sonar.h"
#include "util.h"
#include <fstream.h>
extern void Ping_Sonars();
extern List Buffer;
extern Vehicle Robot;
extern Sonar_Element Sonar_Table[16];
extern Object Current_Parameter;
Config Image;
Config *Current_Path;
Config *Current_Intersect;
Config *Current_Inst;
extern int seq_status;
int Need_Further_Processing;
int TERMINATE = NO;
int TRACE_ROB = NO;
int TRACE_SIM = NO;
extern int TRANSITION;
extern int Path_to_intersect;
fstream Rob_Outfile;
fstream Sim_Outfile;

void Execute_Buffer()
  {
  // Excute Until Terminated
    while (! TERMINATE)
     {
     // Check if Instruction on Buffer is Ready
              if (Buffer.Top().Status() == INCOMPLETE)
                Need_Further_Processing = YES;
              else
                Need_Further_Processing = NO;

     // Detemine Class of Instruction
              switch ( Buffer.Top().Class() )
                {

                  case PARAMETER :
            // Execute Instructions which Change Parameters
                        Execute_Parameter_Commands();
                        break;

                  case STATIONARY :
                        // Execute Instructions when Robot must be Stationary
                        Execute_Stationary_Commands();
              break;

                  case MOVEMENT :
            // Execture Instructions pertaining to Robot Movement
                        Execute_Movement_Commands();
                        break;

                  case ERROR :
                        TERMINATE = YES;
                        break;

                  default:
                        // Buffer is Empty
```

```
                        cout << "Buffer is Empty \n";
                        break;
        }
            if (TRACE_ROB) Execute_Robot_Tracer();
            if (TRACE_SIM) Execute_Sim_Tracer();

     Ping_Sonars();

   // End While Loop
     }

// End Excute_Buffer
  }


void Execute_Robot_Tracer()
 {
 // Write Robot's X, Y and Theta values to a file
   Rob_Outfile << Robot._X   << " " << Robot._Y <<'\n';
 }



void Execute_Sim_Tracer()
 {
 // Write Simulator Data to a file
   Sim_Outfile << Robot._X       << " "
               << Robot._Y       << " "
               << Robot._Theta   << " "
               << Robot._Kappa   << " "
               << Robot._Omega   << "  "
               << Sonar_Table[0] << " "
               << Sonar_Table[1] << " "
               << Sonar_Table[2] << " "
               << Sonar_Table[3] << " "
              << Sonar_Table[4]  << " "  .
              << Sonar_Table[5]  << " "
              << Sonar_Table[6]  << " "
              << Sonar_Table[7]  << " "
              << Sonar_Table[8]  << " "
              << Sonar_Table[9]  << " "
              << Sonar_Table[10] << " "
              << Sonar_Table[11] << '\n';
 }
```

```
void Execute_Parameter_Commands()
 {
 // Determine Parameters to update
   switch (Buffer.Top().Level() )
    {
    // Set S0
            case SET_S0:
              {
              // Execute Set S0
                Execute_Set_S0();
                break;
              }

    // Set Speed
            case SET_SPEED :
              {
              // Execute Set Speed Function
                Execute_Set_Speed();
                break;
              }

            case SET_LACCEL:
              {
              // Execute Set Speed Function
                Execute_Set_L_Accel();
              }

            case TRACE_ROBOT:
              {
              // Execute Trace Robot Function
                Execute_Trace_Robot();
                break;
              }

            case TRACE_SIMULATOR:
              {
              // Execute Trace Simulator Function
                Execute_Trace_Sim();
      }

            case ENABLE_SONAR:
              {
      // Remove command from Buffer
        Current_Parameter = Buffer.Pop();

              // Execute Enable Sonar Group Command
                Execute_Enable_Sonar_Group( Current_Parameter.Variable1());
                break;
      }

            case DISABLE_SONAR:
              {
              // Remove command from Buffer
                Current_Parameter = Buffer.Pop();

              // Execute Disable Sonar Group Function
                Execute_Disable_Sonar_Group(Current_Parameter.Variable1());
                break;
              }

            case SONAR_RANGE_LT:
              {
              // Execute Wait Until Less Then Sonar Range
                Execute_Wait_Until_LT_Sonar_Range();
              }

            case SONAR_RANGE_GT:
              {
```

```
                // Execute Wait Until Greater Then Sonar Range
                  Execute_Wait_Until_GT_Sonar_Range();
                }

              default :
        {
          cout << "ERROR when Excuting Parameters \n" << flush;
                  break;
                }
      // End Switch
      }

  // End Execute_Parameter_Commands
  }



void Execute_Set_S0()
  {
  // Remove command from Buffer
    Current_Parameter = Buffer.Pop();

  // Set Vehicle Speed
    Robot.set_S0(Current_Parameter.Variable1());
  }



void Execute_Set_Speed()
  {
  // Remove command from Buffer
    Current_Parameter = Buffer.Pop();

  // Set Vehicle Speed
    Robot.set_Speed(Current_Parameter.Variable1());
  }

                                  -

void Execute_Set_L_Accel()
  {
  // Remove command from Buffer
    Current_Parameter = Buffer.Pop();

  // Set Vehicle Speed
    Robot.set_L_Accel(Current_Parameter.Variable1());
  }



void Execute_Trace_Robot()
  {
  // Remove Command from Buffer
    Current_Parameter = Buffer.Pop();

  // Open Output file "Vehicle.dat"
    Rob_Outfile.open("Robot.dat", ios::out);

  // Set Trace Flag to Yes
    TRACE_ROB = YES;
  }



void Execute_Trace_Sim()
  {
  // Remove Command from Buffer
    Current_Parameter = Buffer.Pop();
```

```
  // Open Output file "Vehicle.dat"
    Sim_Outfile.open("Sim.dat", ios::out);

  // Set Trace Flag to Yes
    TRACE_SIM = YES;

  }


void Execute_Wait_Until_LT_Sonar_Range()
  {
    double Sonar_Num;
    double Limit;
  //
    Current_Parameter = Buffer.Top();

  // Get Sonar Number
    Sonar_Num = Current_Parameter.Variable1();

  // Get Limit for Sonar Range
    Limit = Current_Parameter.Variable2();

  // Loop Until Sonar Range is Less Then Limit
    if ( (Sonar_Table[Sonar_Num].Range() == 0.0) ||
                (Sonar_Table[Sonar_Num].Range() <= Limit) )
      {
      // Pop Command From Buffer
                Buffer.Pop();
      }
  }


void Execute_Wait_Until_GT_Sonar_Range()
  {
    double Sonar_Num;
    double Limit;
  //
    Current_Parameter = Buffer.Top();

  // Get Sonar Number
    Sonar_Num = Current_Parameter.Variable1();

  // Get Limit for Sonar Range
    Limit = Current_Parameter.Variable2();

  // Loop Until Sonar Range is Greater Then Limit
    if ( (Sonar_Table[Sonar_Num].Range() < 0) &&
        (Sonar_Table[Sonar_Num].Range() >= Limit) )
      {
      // Pop Command From Buffer
                Buffer.Pop();
      }
```

```
void Execute_Stationary_Commands()
 {
 // Determine Level of Instruction
   switch ( Buffer.Top().Level() )
     {

            case SET_ROBOT :
              {
        Execute_Set_Robot();
              break;
       }

            case END :
              {
              Execute_End();
              }

            case ROTATE :
              {
              Execute_Rotate_Robot();
              break;
              }

            case ROTATE_TO :
              {
              Execute_Rotate_Robot_To();
              }

            case STOP_ROBOT :
              {
              Execute_Stop_Robot();
              break;
       }

            default:
       {
                    cout<< "ERROR in STATIONARY LEVEL SWITCH STATEMENT \n" << flush;
                    break;
       }
     }
 }


void Execute_Set_Robot()
 {
 // Get Config and Remove Command from Buffer
   Current_Inst = Buffer.Pop().Command();

 // Set Vehicle Config to Set Robot Config
   Robot = Current_Inst;
 }



void Execute_End()
 {
 // Remove Command from Buffer
   Current_Inst = Buffer.Pop().Command();

 // Set Terminate to Yes
   TERMINATE = YES;
 }
```

```
void Execute_Rotate_Robot()
 {
   double Rotate_Theta;

 // Get Number of Degrees to Rotate
   Rotate_Theta = Buffer.Top().Variable1();

 // Calculate Desired Orientation
   Rotate_Theta = Robot._Theta + Rotate_Theta;

 // Check if Desired Orientation has been Reached
   if ( fabs(Rotate_Theta - Robot._Theta) < .01 )
     {
     // Remove Instruction from Buffer
              Current_Inst = Buffer.Pop().Command();
     }
   else
     {
     // Determine Most Efficient Way to Turn
              if ( Rotate_Theta > Robot._Theta )
                {
        // Rotate Left
                  Robot._Theta += .01;
                }
       else
                {
        // Rotate Right
                  Robot._Theta -= .01;
                }
     }
 }


void Execute_Rotate_Robot_To()
 {
   double Rotate_Theta;

 // Get Desired Orientation
   Rotate_Theta = Buffer.Top().Variable1();

 // Check if at Desired Orientation
   if ( fabs(Rotate_Theta - Robot._Theta) < .01 )
     {
     // Remove Instruction from Buffer
              Current_Inst = Buffer.Pop().Command();
     }
   else
     {
     // Determine Most Efficient Way to Turn
              if ( Rotate_Theta > Robot._Theta )
                {
        // Rotate Left
                  Robot._Theta += .01;
                }
       else
                {
                // Rotate Right
                  Robot._Theta -= .01;
                }
     }
 }
```

```
void Execute_Stop_Robot()
 {
 // Current_Inst = Buffer.Pop().Command();
   TERMINATE = YES;
 }


void Execute_Movement_Commands()
 {
 // Determine Level of Instruction
   switch ( Buffer.Top().Level() )
    {
    // Intersect
            case INTERSECT :
              {
         Execute_Intersection_Command();
              break;
       }

            case FULL_PATH :
              {
         Execute_Full_Path_Command();
              break;
       }

            case PARTIAL_PATH :
              {
                Execute_Partial_Path_Command();
                break;
       }

            case CUBIC_SPIRAL:
       {
              break;
             }

            default :
       {
              cout << "ERROR with Movement Level \n" << flush;
         break;
       }
    }
 }


void Execute_Intersection_Command()
 {
   Config Current_Intersect;
   int TRANSITION;
//   Config Image;
   double S0_Mult;

 // Set Intersect to Value of Instruction
   Current_Intersect = Buffer.Top().Command();

 // Get S0 Multiplier for Transition
   S0_Mult = Buffer.Top().Variable1();

 // Calculate Image to Current Path
   Image = Current_Path->image();

 // Update Vehicle Configuration to Move onto Current Path
   Update_config(Current_Path, Image);

 // Calculate if at Transition Point
   TRANSITION = Image.Transition(Current_Intersect, S0_Mult);
```

```
  // Check if Vehicle should Transition to New Path
    if (TRANSITION)
      {
      // Remove Intersect Instruction from Buffer
              Current_Intersect = Buffer.Pop().Command();

      }
  }


void Execute_Full_Path_Command()
  {
  // Check if Path is Ready
    if ( Need_Further_Processing )
      {
      // Not Ready Set Current Path to Value of Instruction
        Current_Path = Buffer.Top().Command();
      }
    else
      {
      // Ready: Remove Instruction from Buffer
        Current_Path = Buffer.Pop().Command();
      }

  // Calculate Image to Current Path
    Image = Current_Path->image();

  // Update Vehicle Configuration to Move onto Current Path
    Update_config(Current_Path, Image);

  }


void Execute_Partial_Path_Command()
  {
    int TRANSITION = NO;
              TERMINATE = YES;
  // Check if Time to Transition
    if (! TRANSITION)
      {
      // Not Ready: Set Current Path to Value of Instruction
              Current_Path = Buffer.Top().Command();

      // Calculate Image to Current Path
              Image = Current_Path->image();

      // Update Vehicle Configuration to Move onto Current Path
              Update_config(Current_Path, Image);

      // Check if Time to Transition to new Path
              TRANSITION = Image.Complete(Current_Path);
      }
    else
      {
      // Ready: Remove Instruction from Buffer
              Current_Path = Buffer.Pop().Command();

      // Reset Transition Flag to No
//            TRANSITION = NO;
               TERMINATE = YES;
      }

  }
```

```
void Execute_Cubic_Spiral_Command()
 {
   Current_Path = Buffer.Top().Command();
   Image = Current_Path->image();
   Update_config(Current_Path, Image);
   TERMINATE = YES;

 }
```

LIST OF REFERENCES

[1] Abresch, R.J., "Path Tracking Using Simple Planar Curves," Master Thesis, Naval Postgraduate school, Monterey, California, March 1992.

[2] Shapiro, J.S., "A C++ Toolkit" Prentice-Hall, Inc., 1991, pp 1-8.

[3] MacPherson, D., Kanayama, Y., "Theory and Experiments on Autonomous Vehicle Navigation with Real-Time Odometry Error Correction," Technical Report of the Department of Computer Science, Naval Postgraduate School, Monterey, California, September 1992.

[4] Kanayama, Y., MacPherson, D., "Two Dimensional Transformations and its Applications to Vehicle Motion Control and Analysis," International IEEE Conference on Robotics and Automation, Atlanta, Georgia, to appear.

[5] Alexlander, J.A., "Motion Control and Obstacle Avoidance for a Autonomous Vehicle using Simple Planar Curves," Master Thesis, Naval Postgraduate School, Monterey, California, March 1993.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943	2
3.	Chairman, Computer Science Department Code CS Naval Postgraduate School Monterey, CA 93943	1
4.	Professor Yutaka Kanayama Code CSKa Naval Postgraduate School Monterey, CA 93943	1
5.	Dr. Michael J. Zyda Code CSZk Naval Postgraduate School Monterey, CA 93943	1
6.	LCDR David L. MacPherson Code CSPM Naval Postgraduate School Monterey, CA 93943	1
7.	LT Carl J. Grim 300 Glenwood Circle #406 Monterey, CA 93940	2